Efficient Version Space Algorithms for Human-in-the-loop Model Development

Luciano Di Palma, Yanlei Diao, Anna Liu

▶ **To cite this version:**

Luciano Di Palma, Yanlei Diao, Anna Liu. Efficient Version Space Algorithms for Human-in-the-loop Model Development. ACM Transactions on Knowledge Discovery from Data (TKDD), 2024, 18 (3), pp.1-49. 10.1145/3637443 . hal-04414855

HAL Id: hal-04414855
https://inria.hal.science/hal-04414855v1

Submitted on 24 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Efficient Version Space Algorithms for Human-in-the-Loop Model Development

LUCIANO DI PALMA, Ecole Polytechnique, France
YANLEI DIAO, Ecole Polytechnique, France
ANNA LIU, University of Massachusetts Amherst, United States

When active learning (AL) is applied to help users develop a model on a large dataset through interactively presenting data instances for labeling, existing AL techniques often suffer from two main drawbacks: First, to reach high accuracy they may require the user to label hundreds of data instances, which is an onerous task for the user. Second, retrieving the next instance to label from a large dataset can be time-consuming, making it incompatible with the interactive nature of the human exploration process. To address these issues, we introduce a novel version-space-based active learner for kernel classifiers, which possesses strong theoretical guarantees on performance and efficient implementation in time and space. In addition, by leveraging additional insights obtained in the user labeling process, we can factorize the version space to perform active learning in a set of subspaces, which further reduces the user labeling effort. Evaluation results show that our algorithms significantly outperform state-of-the-art version space strategies, as well as a recent factorization-aware algorithm, for model development over large data sets.

CCS Concepts: • **Computing methodologies** → **Active learning settings**.

Additional Key Words and Phrases: active learning, version space, kernel classifiers, hit-and-run, factorization

1 INTRODUCTION

Active learning [15, 33, 37, 38] has been studied extensively to address the problem of learning classification models with limited training data. An active learner can examine the current classification model and develop a series of inquiries from the data source to obtain new labeled instances that increase classification accuracy as fast as possible, hence reducing the overall burden of sample acquisition and labeling.

This line of work has gained increased interest in a recent industry trend known as "Machine Learning for Everyone". In this trend, IT companies deliver cloud platforms such as Amazon SageMaker[1] and Google AutoML[2] to help users develop machine learning models from their data sets with minimum effort. However, a key concern with machine learning models is their need for large volumes of high-quality training data: although data itself is abundant in most applications, annotated data is often much scarcer. Current industry solutions for this "lack of labeled data" problem are often limited to crowdsourcing or manual labeling by dedicated IT teams, but these approaches are unlikely to meet the needs of millions of data sets to be hosted on these platforms, especially when domain knowledge is required (e.g., to label medical data).

More recently, one line of work that shows promising results in this direction is Interactive Data Exploration (IDE) [10, 11, 18, 23], which is a new data exploration framework that brings active learning to bear on the new process of model learning. In this setting, the user aims to build a

[1]https://aws.amazon.com/sagemaker/
[2]https://cloud.google.com/automl/

Authors' addresses: Luciano Di Palma, luciano.di-palma@polytechnique.edu, Ecole Polytechnique, Palaiseau, France, 91120; Yanlei Diao, yanlei.diao@polytechnique.edu, Ecole Polytechnique, Palaiseau, France, 91120; Anna Liu, anna@math.umass.edu, University of Massachusetts Amherst, Amherst, MA, United States, 01003.

©
ACM 1556-4681/2023/12-ART1
https://doi.org/10.1145/1122445.1122456

classification model over a data set with minimal effort from no or very few labeled instances. Active learning is applied to select a minimum sequence of data instances for the user to label in order to derive an accurate model while, at the same time, offering interactive performance in presenting the next data instance for the user to review and label. As such, IDE presents a more challenging setting for active learning than traditional model learning: besides minimizing labeling effort, it further poses a performance requirement when scanning the underlying data set to identify the best data instances for labeling next.

However, existing active learning techniques often fail to provide satisfactory performance when such models need to be built over large data sets. For example, our evaluation results show that, on a Sloan Digital Sky Survey (SDSS) data set of 1.9 million data instances, existing active learning techniques [15, 33, 37] and a recent active-learning-based data exploration system [18] require the user to label over 200 instances to learn a classification model on 6 attributes with an F-score of 80%.[3] Asking the user to label hundreds of instances to achieve high accuracy is undesirable, referred to as the *slow start* problem in this paper. In our work, we aim to devise new techniques to overcome the slow start problem and expedite convergence while providing interactive performance for user labeling. Our design of new techniques embodies two main ideas: *Version Space algorithms* and *Factorization*, which we discuss in more detail below.

1.1 Version Space Algorithms

First, we would like to leverage Version Space (VS) algorithms [7, 14, 33, 37] because they present a strong theoretical foundation for convergence. These algorithms model all possible configurations of a classifier that can correctly classify the current set of labeled data as a set of hypotheses forming a *version space* $\mathcal{V}$, and aim to find the next instance such that its acquired label will enable $\mathcal{V}$ to be reduced. The best-known example is the Generalized Binary Search (GBS) algorithm [7, 14] which, among all unlabeled instances, looks for the one whose label allows $\mathcal{V}$ to be reduced by half or most close to that. It is shown theoretically to have near-optimal performance for convergence.

Despite its theoretical advantages, the GBS algorithm is prohibitively expensive to run in practice due to the exponential size of the version space [32]. In the literature, several approximations were proposed to counter this limitation, each offering a different trade-off among convergence speed, time per iteration, and theoretical guarantees. Consider popular version space algorithms, including Simple Margin [37], Query-by-Disagreement (QBD) [33], Kernel Query-by-Committee (KQBC) [12], and ALuMA [15]. On one hand, Simple Margin and QBD are fast, rough approximations of the GBS strategy and, consequently, suffer from suboptimal performance. On the other hand, KQBC and ALuMA can better approximate the bisection rule via a sampling procedure, but these methods can be very time-consuming. For example, ALuMA runs at every iteration of data exploration a hit-and-run sampling procedure with thousands of steps to select a single hypothesis and repeats this process hundreds of times to generate a sufficiently large sample of classifiers. Finally, among all of the above techniques, ALuMA is the only one with provable theoretical guarantees on performance.

Given the limitations of existing VS algorithms, in this work, we seek to design a new version space framework by exploring new theoretical analysis and optimization to meet the fast convergence and interactive performance requirements in the setting of efficient model development.

[3]F-score is a better measure for exploring a large data set than classification error given that the user interest, i.e., the positive class, often covers a (very) small fraction of the data set; classifying all instances to the negative class has low classification error, but is useless for retrieving relevant data, hence poor F-score.

1.2 Factorization

To make version space algorithms practical for large data sets, our second idea is to augment them with additional insights obtained in the user labeling process. In particular, we observe that, often, when a user labels a data instance, the decision-making process can be broken into a set of simpler questions whose answers can be combined to derive the final answer. For example, when a customer decides whether a car model is of interest, he may have a set of questions in mind:

- Q_1: Is the gas mileage good enough?
- Q_2: Is the vehicle spacious enough?
- Q_3: Is the color a preferred one?

While the user may have a high-level intuition about how each question is related to a subset of attributes, he may not be able to specify these questions precisely. In fact, doing so would require precise knowledge of the relationship between the attributes composing each question, including any constants and thresholds used. For example, the user may know a vehicle's space is a function of the body type, length, width, and height, but he probably does not know how these values are connected or which volume threshold precisely captures his interest.

The above insight allows us to design a new version space algorithm that leverages the high-level intuition a user has for breaking the decision-making process, formally called a *factorization structure*, to combat the slow start problem.

1.3 Summary of Contributions

More specifically, we make the following contributions in this work:

(1) *A New Theoretical Framework for Version Space (VS) Algorithms over Kernel Classifiers* (Section 3): We propose a new theoretical framework that allows for an efficient implementation of the Generalized Binary Search [14] strategy over kernel classifiers. Compared to previous works [12, 15], our framework offers both strong theoretical guarantees on performance and efficient implementation in time and space. Our key techniques include a dimensionality reduction theorem that restricts the kernel matrix computations to the set of labeled points and a Cholesky-decomposition-based method for efficient estimation of the version space reduction of any sample. We also prove generalization error bounds on accuracy and F-score, enabling our techniques to run over a sample from the original large data set with minimal performance loss.

(2) *Implementation and Optimizations* (Section 4): Based on our theoretical results, we propose an optimized version space algorithm called OptVS. Similar to the works of Gilad-Bachrach et al. [12] and Gonen et al. [15], OptVS uses the hit-and-run algorithm [25] for sampling the version space. However, hit-and-run may require thousands of iterations to output a high-quality sample, which can incur a high time cost. To reduce the cost, we develop a range of sampling optimizations to improve the sample quality and running time. In particular, we provide a highly efficient version of the *rounding* technique [24] for improving the sample quality from the version space.

(3) *A Factorized Version Space Algorithm* (Section 5): Additionally, we propose a new algorithm that leverages the factorization structure indicated by the user to create subspaces and factorizes the version space accordingly to perform active learning in the subspaces. Compared to recent work [18] that also used factorization for active learning, our work explores it in the new setting of VS algorithms and eliminates the strong assumptions made, such as convexity of user interest patterns, resulting in significant performance improvement while increasing the applicability in real-world problems. We also provide theoretical results on the

optimality of our factorized VS algorithm, as well as optimizations for dealing with subspaces of categorical variables.

Using two real-world data sets and a large suite of user interest patterns, our evaluation results (Section 6) show that (1) for lower dimensional problems, our optimized VS algorithm, without factorization, already outperforms existing VS algorithms, including Simple Margin [37], Query-by-Disagreement [33], and ALuMA [15]; (2) for higher-dimensional problems, our factorized VS algorithm outperforms non-factorized active learners [15, 33, 37], as well as DSM [18], a factorization-aware algorithm, often by a wide margin while maintaining interactive speed. For example, for a complex user interest pattern tested, our algorithm achieves an F-score of over 80% after 100 labeling iterations, while DSM is still at 40% and other VS algorithms at 10% or lower.

2 RELATED WORK

In this section, we survey the most relevant works for each of our proposed contributions, including version space algorithms and interactive data exploration systems.

2.1 Version Space Algorithms

We first provide an overview of active learning techniques, in particular, version space algorithms.

Active Learning. Settles [33] and Monarch [29] provide an overview of the most relevant results in the active learning (AL) literature. Our work focuses on a particular setting called pool-based active learning, in which a classification model is built over a large data set by relying on user feedback over targeted data samples. More precisely, based on the current classification model, the active learning algorithm can inspect the data pool and extract the data example that, if labeled, will result in the fastest increase in classification accuracy. In doing so, active learners can build an accurate classifier with few labeled examples and, consequently, minimal user effort.

In practice, AL is usually employed in domains where labeling data is expensive and labeled data sets are scarce. For example, recent fields of application include biomedical image segmentation [40], health monitoring [5], and crisis management [30]. Active learning has also been applied in crowd-sourcing scenarios [16, 36], where the labeling task is outsourced to multiple annotators (which are unlikely to be domain experts). In our work, we aim to apply active learning, in particular the version space algorithms, in the settings of human-in-the-loop model development (i.e., an application user is actively engaged in visualization and labeling to build a model efficiently).

Version Space Algorithms. Version space (VS) algorithms are a particular class of active learning strategies leveraging the extra information found in the set of classifiers (hypotheses) under consideration. More precisely, these procedures start with a hypothesis set $\mathcal{H}$ and define the version space $\mathcal{V}$ as the subset of classifiers $h \in \mathcal{H}$ consistent with the user-labeled data, that is, $h(x) = y$ for all labeled points (x, y). With this, VS algorithms aim to reduce $\mathcal{V}$ as quickly as possible until we are left with a single hypothesis achieving perfect classification. Various strategies have been developed to estimate the VS reduction of each sample, each offering a different trade-off in terms of performance, efficiency, and theoretical guarantees. Below, we give an overview of the most relevant algorithms in the literature:

- *Generalized Binary Search* [7, 14]: Also called the *Version Space Bisection* rule, this algorithm searches for a point x for which the classifiers in $\mathcal{V}$ disagree the most; that is, the sets $\mathcal{V}^{x,y} = \{h \in \mathcal{V} : h(x) = y\}$ have roughly the same size for all possible labels y. The main advantage of this strategy is its strong theoretical guarantees on convergence: on average, the expected number of iterations needed to reach 100% accuracy is at most a log factor higher than the optimal algorithm. However, implementing the bisection rule is prohibitively expensive: for each instance x in the data set, one must evaluate $h(x)$ for each hypothesis

h in the version space, which is exponential in size $O(m^D)$, where m is the size of the data set and D is the VC dimension [32]. Due to this limitation, several approximations of the bisection rule have been introduced in the literature.

- *Simple Margin* [37]: Simple Margin is a rough approximation of the bisection rule for SVM classifiers, leveraging the heuristic that data points close to the SVM's decision boundary closely bisect the version space $\mathcal{V}$. However, this method can suffer from suboptimal performance in practice, especially when $\mathcal{V}$ is very asymmetric.
- *Kernel Query-by-Committee* (KQBC) [12] and *Query-by-Disagreement* (QBD) [33]: These algorithms approximate the bisection rule by constructing two representative hypotheses in the version space and then select any data point for which these two classifiers disagree. In particular, KQBC samples two random hypotheses from the version space, while QBD trains one positive and one negatively biased classifier. Again, these methods can suffer from suboptimal performance since the selected point is only guaranteed to "cut" $\mathcal{V}$, but possibly not by a significant amount.
- *ALuMA* [15]: ALuMA is an application of the Generalized Binary Search [14] algorithm for linear classifiers and uses a sampling technique for estimating the version space reduction of any sample. It can also support kernels via a preprocessing step, but this procedure can be very expensive to run. Additionally, ALuMA was shown to outperform several other version space algorithms, including Simple Margin and Query-by-Committee [34]. Hence, we use ALuMA as a baseline version space algorithm in this work.

More recently, version-space-based techniques have also been successfully applied to a broader range of scenarios, such as cost-sensitive classification [21] and batch-mode active learning [6]. In this work, we focus on how version space techniques can be applied to the interactive model development problem.

2.2 Data Exploration and Model Development

Next, we consider other areas of interactive model development in the literature, including recent human-in-the-loop data exploration systems.

Data Programming under Weak Supervision. One form of interactive model development is the *data programming* framework introduced by Snorkel [2, 31], which has gained attraction in recent years. In this framework, an expert user writes a set of *labeling functions* representing simple heuristics used for labeling data instances. By treating these labeling functions as weak learners, Snorkel can build an accurate classifier without having the user manually label any data instance. In more recent work, a data programming system called Snuba [39] was designed to automatically generate labeling functions from an initial labeled data set, leading to improved classification accuracy and further reduction of the user manual effort. However, this system still requires an initial labeled set on the order of hundreds to thousands of labeled instances, which is nontrivial to obtain, especially in the new trend of "Machine Learning for Everyone" where an everyday user may start with no labeled data at all.

Interactive Data Exploration. In the interactive data exploration domain, the main objective is to design a system that guides the user towards discovering relevant data points in a large data set. One recent line of work is "explore-by-example" systems [10, 11, 18, 23], which leverage active learning techniques to construct an accurate model of the user interest from a small set of user-labeled data points. These human-in-the-loop systems require minimizing both the user labeling effort and the running time to find a new data point for labeling in each iteration. In particular, recent work [18] is shown to outperform prior work [10, 11] via the following technique:

Dual-Space Model (DSM) [18]: In this work, the user interest pattern is assumed to form a convex object in the data space. By leveraging this property, this work proposes a dual-space model (DSM) that builds a polytope model of the user interest on top of any active learning strategy. In particular, this polytope model partitions the data space into three segments: the *positive region*, where points are guaranteed to be interesting, the *negative region*, where points are guaranteed to be uninteresting, and the remaining *unknown region*. In doing so, DSM can use the polytope model to automatically label most of the data points selected by the active learner and, consequently, significantly reduce the user labeling effort.

In addition, DSM can also leverage a *factorization structure* provided by the user in a limited form, i.e., when the user pattern is a conjunction of convex or concave sub-patterns. If such conditions are met, the system then factorizes the data space into low-dimensional subspaces and runs the dual-space model in each subspace. In doing so, the original complex exploration process is broken into several simpler tasks, thus significantly improving the performance of their system.

Previous Work on Factorization: We also compare the contributions of this paper with our previous work [9]. In [9], our main contribution was to apply the idea of factorization to existing (unoptimized) version space algorithms: Our technique first decomposes the version space into several *version subspaces* using the factorization structure provided by the user and then applies an extended version of the bisection strategy to propose new points for the user to review and label.

The work described in this paper significantly extends [9] by proposing a new framework for efficient version space algorithms over kernel classifiers, optimizations to expedite the sampling procedure of the version space, and extension of our version space algorithms with factorization. More specifically, our major extensions include:

- First, we introduce a novel *theoretical framework* for version space algorithms over kernel classifiers (Section 3). Compared to [9], our new techniques possess strong theoretical guarantees on performance while enabling several optimizations, including a dimensionality reduction theorem that restricts the kernel computations to the set of labeled points, a Cholesky decomposition-based technique for efficiently estimating the version space reduction of any sample, and generalization error bounds that enable our techniques to run over a sample from the original data with minimal performance loss.
- We have also included *new optimizations* for efficiently sampling the version space (Sections 4.1 and 4.3), including an ellipsoid caching optimization and a novel rounding algorithm that is both efficient and numerically stable.
- The section on factorization (Section 5) has also been extended with two new results (Sections 5.5 and 5.6): a new factorization strategy called "product loss", which is shown in our evaluation to outperform the other strategies, as well as optimizations for categorical subspaces.
- Finally, our experimental results (Section 6) have been further extended, including 18 new query results over a new data set.

3 GENERALIZED BINARY SEARCH OVER KERNEL CLASSIFIERS

In this section, we address the problem of how to efficiently realize the Generalized Binary Search (GBS) algorithm for kernel classifiers. Although a few works [12, 15] have also attempted to apply version-space-based techniques to kernel classifiers, their approaches fail to scale to large data sets.

In Gonen et al. [15], they have developed a novel algorithm, ALuMA, for applying the GBS strategy over linear classifiers. This algorithm comes with several theoretical results on convergence speed and label complexity guarantees. ALuMA also supports an extension to kernel classifiers by first running a preprocessing step over the data. However, this procedure has two major drawbacks:

first, it requires computing the kernel matrix (and its Cholesky decomposition) over all data points, which is time and memory inefficient for large data sets; second, this preprocessing step requires an upper bound on the total Hinge loss of the best separator, which is usually not known in practice.

In Gilad-Bachrach et al. [12], they proposed an extension of the Query-by-Committee [34] algorithm to kernel classifiers called Kernel Query-by-Committee (KQBC). They also demonstrated a dimensionality reduction theorem that avoids computing the kernel matrix over the entire data set, improving the algorithm's efficiency. However, estimating whether a data point "cuts" the version space or not requires running a separate sampling procedure, which can become particularly expensive in the pool-based setting. Another drawback of this work is the lack of theoretical guarantees on convergence speed.

In what follows, we introduce a new version-space technique over kernel classifiers. Compared to previous work, our approach shares the same near-optimal guarantees on convergence as GBS while still allowing for an efficient implementation in both time and space similar to KQBC, thus being compatible with the interactive data exploration scenario.

3.1 Generalized Binary Search Algorithm Overview

Let $\mathcal{X} = \{x_i\}_{i=1}^N$ be a collection of unlabeled data points, and let $y_i \in \mathcal{Y}$ represent the unknown label of x_i. The set $\mathcal{Y}$ is called the *label space*, which is assumed to be finite. The user interest pattern can be modeled by an *hypothesis function* $h : \mathcal{X} \to \mathcal{Y}$, with h belonging to some *hypothesis space* $\mathcal{H}$. This space is assumed to satisfy the following condition:

Definition 3.1 (Realizability or Separability). There is a classifier $h^* \in \mathcal{H}$ matching the user preference, that is, achieving zero classification error.

With this, our objective is to identify the hypothesis h^* while minimizing the number of instances labeled by the user.

In the above formulation, it is possible to have more than one hypothesis $h^* \in \mathcal{H}$ matching the user interest. One well-known solution [7, 14, 15] for this ambiguity problem is to define an equivalence relation over $\mathcal{H}$ which identifies any two hypotheses having the same predictions over $\mathcal{X}$:

$$h \sim h' \iff h(x_i) = h'(x_i),\ \forall x_i \in \mathcal{X} \tag{1}$$

With this, our original problem becomes finding the equivalence class $[h^*]$, which is unique. Additionally, we assume a known probability distribution $\pi([h])$ over the set of equivalence classes $\hat{\mathcal{H}} := \mathcal{H}/\sim$, representing our prior knowledge over which hypotheses are more likely to match the user's preferences. In the absence of prior knowledge, a uniform prior can be assumed. In what follows, to simplify notation, we will drop the brackets when referring to equivalence classes $[h]$ whenever it is clear from the context.

The *version space* [27] is the set of all hypotheses consistent with a labeled set $\mathcal{L}$: $\mathcal{V} = \{h \in \hat{\mathcal{H}} : h(x) = y,\ \forall (x, y) \in \mathcal{L}\}$. Additionally, let's introduce the shorthand $\mathcal{V}^{x,y} = \mathcal{V} \cap \{h \in \hat{\mathcal{H}} : h(x) = y\}$. Then, the *version space bisection rule* searches for the point x such that all the sets $\mathcal{V}^{x,y}$ have approximately the same probability mass:

Definition 3.2 (Generalized Binary Search - GBS). Let $\mathcal{V}$ denote the version space at any iteration of the active learning loop. For any $(x, y) \in \mathcal{X} \times \mathcal{Y}$, we define the **cut probabilities**:

$$p_{x,y} = \mathbb{P}(h(x) = y \mid h \in \mathcal{V}) = \frac{\pi(\mathcal{V}^{x,y})}{\pi(\mathcal{V})} \tag{2}$$

Symbol	Description
$\mathcal{X}$	Set of unlabeled data points $\{x_i\}_{i=1}^N$
y_i	The (unknown) label for x_i taking values in a finite set $\mathcal{Y}$
$\mathcal{L}$	Set of data points labeled by the user and their labels at any iteration of the AL process
$\mathcal{U}$	Set of points that have yet to be seen by the user at any iteration of the AL process
$\mathcal{H}$	Set of hypotheses (e.g., classifiers) $\{h : \mathcal{X} \to \mathcal{Y}\}$ mapping data points to labels
h	A classifier $h \in \mathcal{H}$
h^*	An optimal hypothesis in $\mathcal{H}$ achieving zero classification error
$\hat{\mathcal{H}} = \mathcal{H}/\sim$	Set of equivalence classes $[h]$ over $\mathcal{H}$ as defined in Eq. (1)
$\mathcal{V}$	The Version Space (VS), that is, the subset of hypotheses in $\hat{\mathcal{H}}$ matching the labeled set $\mathcal{L}$
π	Probability mass over the version space representing our prior information on h^*
$p_{x,y}$	Cut probabilities, as defined by Eq. (2)
Ω	Set of allowed parameters for kernel classifiers
k	A positive-definite kernel function
K and L	The kernel matrix over $\mathcal{X}$ and its Cholesky decomposition
ℓ_j	The partial Cholesky factors associated with x_j, as defined in Eq. (7).
W_t	Reduced version space, as defined in Eq. (8)
$\epsilon(h, S)$	The error of a classifier h over a subset $S \subset \mathcal{X}$
MV^π	The Majority Vote function, as defined in Section (3.4.1)

Table 1. Notation table for Section 3.

Then, the GBS strategy selects any unlabeled point $x^* \in \mathcal{U}$ satisfying

$$x^* \in \arg\max_{x \in \mathcal{U}} 1 - \sum_{y \in \mathcal{Y}} p_{x,y}^2$$

One main advantage of the GBS strategy is it enjoys near-optimal performance guarantees [14]. Let $cost(\mathcal{A})$ denote the average number of queries an active learner $\mathcal{A}$ takes to identify a random hypothesis drawn from π. Then, the GBS algorithm satisfies

$$cost(GBS) \leq OPT \cdot \left(1 + \log \frac{1}{\min_h \pi(h)}\right)^2 \tag{3}$$

with $OPT = \min_{\mathcal{A}} cost(\mathcal{A})$. In other words, when simply considering the number of iterations until convergence, the GBS strategy differs from the optimum by at most a log factor. In particular, in the case were π is uniform and $\mathcal{H}$ has finite VC-dimension D, Sauer's Lemma [32] allows us to write $\log(1/\min_h \pi(h)) = \log|\hat{\mathcal{H}}| = O(D \log N)$, which discloses a logarithmic dependency on the number of data points N.

3.2 Kernel Classifiers

With the above, we turn to the problem of efficiently realizing the GBS strategy over kernel classifiers. First, let's restrict ourselves to the *binary case* and set $\mathcal{Y} = \{\pm 1\}$. Additionally, let $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be a *positive-definite kernel function*,[4] with corresponding *feature map* $\phi : \mathbb{R}^d \to \mathcal{F}$. Our objective is to apply the GBS algorithm to the set of linear classifiers over $\mathcal{F}$

$$\mathcal{H} = \{h_{b,f} : h_{b,f}(x) = \text{sign } b + \langle \phi(x), f \rangle_{\mathcal{F}}, \text{ for } (b, f) \in \Omega\}$$

[4]Any kernel k can be made positive-definite by adding a small perturbation: $k'(x, y) = k(x, y) + \lambda \mathbb{1}(x = y)$, with $\lambda > 0$

in which $\Omega = \{(b, f) \in \mathbb{R} \times \mathcal{F} : b^2 + \|f\|_{\mathcal{F}}^2 \leq 1\}$ is the parameter set. The constraint $b^2 + \|f\|_{\mathcal{F}}^2 \leq 1$ can be included without any loss in generality since $h_{b,f} \sim h_{rb,rf}$ for any $r > 0$.

One major problem of these classifiers is the high dimensionality of $\mathcal{F}$, being even infinite-dimensional in certain cases [28]. However, it is always possible to restrict the set of parameters to a finite-dimensional subspace of $\mathcal{F}$ without loss, which is the subject of our next theorem:

Lemma 3.3. *Let $S = span(\phi(x_1), \ldots, \phi(x_N))$ and let $f|_S$ denote the orthogonal projection of $f \in \mathcal{F}$ onto S. Then, for all $f \in \mathcal{F}$ and all $x_i \in \mathcal{X}$ we have:*

$$\langle f, \phi(x_i)\rangle_{\mathcal{F}} = \langle f|_S, \phi(x_i)\rangle_{\mathcal{F}}$$

In particular, this implies that $h_{b,f} \sim h_{b,f|_S}$ for all $(b, f) \in \mathbb{R} \times \mathcal{F}$.

With this, we can restrict ourselves to the set of parameters of the form $f = \sum_{i=1}^{N} \alpha_i \phi(x_i)$, which leads to the following definition:

Definition 3.4 (Kernel Classifiers). Let K be the $N \times N$ kernel matrix $K_{ij} = k(x_i, x_j)$. The set of *kernel classifiers* is defined as

$$\mathcal{H}_\alpha = \{h_{b,\alpha} : h_{b,\alpha}(x) = \text{sign } b + \sum_{i=1}^{N} \alpha_i k(x_i, x), \text{ for } (b, \alpha) \in \Omega_\alpha\}$$

where $\Omega_\alpha = \{(b, \alpha) \in \mathbb{R} \times \mathbb{R}^N : b^2 + \alpha^T K \alpha \leq 1\}$ is the corresponding set of parameters.

With this, we are ready to define the GBS strategy over kernel classifiers:

Definition 3.5 (Kernel GBS). Let $\mathcal{H}_\alpha$ be the set of kernel classifiers and let $\pi_{b,\alpha}$ be a prior over Ω_α. In addition, let us define the *kernel version space* $\mathcal{V}_\alpha$ as the set:

$$\mathcal{V}_\alpha = \{(b, \alpha) \in \Omega_\alpha : y_i(b + \alpha^T K_i) \geq 0, \forall (x_i, y_i) \in \mathcal{L}\}$$

where K_i is the row of the kernel matrix K corresponding to x_i. Then, the *kernel GBS strategy* selects the unlabeled point x^* satisfying:

$$x^* \in \arg\max_{x \in \mathcal{U}} 1 - \sum_{y \in \mathcal{Y}} p_{x,y}^2$$

where $p_{x_j,y} = \mathbb{P}_{(b,\alpha)\sim\pi}\left(y\,(b + \alpha^T K_j) \geq 0 \,\middle|\, (b, \alpha) \in \mathcal{V}_\alpha\right)$.

In particular, it is simple to show that our kernel GBS strategy possesses similar near-optimality guarantees as the general GBS algorithm:

Theorem 3.6. *Let GBS_{kernel} be the kernel GBS strategy as defined above. Then, this strategy satisfies*

$$cost(GBS_{kernel}) \leq OPT \cdot \left(1 + \log \frac{1}{\min_h \tilde{\pi}(h)}\right)^2$$

where $OPT = \min_{\mathcal{A}} cost(\mathcal{A})$ and $\tilde{\pi}(h) = \mathbb{P}_{(b,\alpha)\sim\pi}(h_{b,\alpha} \sim h)$.

In what follows, we focus on how to efficiently compute the probabilities $p_{x_j,y}$.

3.3 Dimensionality Reduction

One major problem in computing the cut probabilities $p_{x_j,y}$ is the high-dimensionality of Ω_α: since the number of data points N is usually very large, each $p_{x_j,y}$ will be very expensive to estimate. Below, we show a method for reducing the dimensionality from $N + 1$ to $|\mathcal{L}| + 2$, a more tractable range of values.

We start with a simplifying assumption:

Assumption for this section: In what follows, we assume the labeled set $\mathcal{L}$ to be composed of the first t data points $\{x_1, \ldots, x_t\}$, which is always possible to do since it amounts to a simple re-indexing of data points. Besides, we assume a uniform prior $\pi_{b,\alpha}$ over the set of parameters Ω_α of kernel classifiers.

Our first step is to consider the Cholesky decomposition $K = LL^T$, with L being lower-triangular. By defining the change of variables $\beta = L^T\alpha$, the cut probabilities can be slightly simplified to

$$p_{x_j,y} = \mathbb{P}\left(y\,(b + \beta^T L_j) \geq 0\right)$$

where L_j is the j-th row of L. Besides, since L^T is invertible and we are assuming a uniform prior over Ω_α, (b, β) must also follow the uniform distribution over the set

$$\mathcal{V}_t = \{(b, \beta) \in \Omega_\beta : y_i(b + \beta^T L_i) \geq 0,\ 1 \leq i \leq t\} \tag{4}$$

where Ω_β is the unit ball in $\mathbb{R}^{N+1}$. With this, we can apply the following dimensionality reduction lemma, which is an extension of Theorem 1 from Gilad-Bachrach et al. [12]:

Lemma 3.7. *Define $S_j = \mathbb{R} \times span(L_1, \ldots, L_t, L_j)$, and let (b', β') be drawn uniformly over $\mathcal{V}_t \cap S_j$. Then, the cut probabilities $p_{x_j,y}$ satisfy*

$$p_{x_j,y} = \mathbb{P}\left(y\,(b' + \beta'^T L_j) \geq 0\right) \tag{5}$$

With this result, the dimensionality is effectively reduced from $N + 1$ to $t + 2$, and algorithms should be much more efficient to run. However, there is still a slight problem: the vectors L_j live in an N-dimensional space and thus can be costly to compute. To fix this, we first construct an orthonormal basis for S_j:

Lemma 3.8. *Let's denote by L_j^k the entry of L at row j and column k. Then, for any $j > t$ the set*

$$\left\{\vec{e_1}, \vec{e_2}, \ldots, \vec{e_{t+1}}, \frac{\vec{T_j}}{\|\vec{T_j}\|}\right\}$$

is an orthonormal basis for S_j, where $\vec{e_i}$ is the i-th canonical vector in $\mathbb{R}^{N+1}$ and $\vec{T_j} = (\vec{0}_{t+1}, L_j^{t+1}, \ldots, L_j^N)$.

Now we have an orthonormal basis, we can represent any vector $(b', \beta') \in S_j$ as a sum

$$(b', \beta') = \sum_{i=1}^{t+1} w_i \vec{e_i} + w_{t+2} \frac{\vec{T_j}}{\|\vec{T_j}\|} \tag{6}$$

and $w \in \mathbb{R}^{t+2}$ contains the orthonormal basis coefficients. By replacing this representation in Equations (4) and (5), we obtain our main result:

Definition 3.9 (Partial Cholesky Factor). Let $\mathcal{L} = \{(x_1, y_1), \ldots, (x_t, y_t)\}$ be the labeled set, and let $K = LL^T$ be the Cholesky decomposition of the $N \times N$ kernel matrix over all data points. We define the *partial Cholesky factor* $\ell_i \in \mathbb{R}^{t+2}$ as

$$\ell_i = \left(1, L_i^1, \ldots, L_i^t, \sqrt{K_{ii} - \sum_{k=1}^{t} (L_i^k)^2}\right) \tag{7}$$

In particular, we note that $\ell_i^r = 0$ whenever $i \leq \min(t, r-2)$.

Theorem 3.10. *Let w be a random vector drawn uniformly over the set*

$$W_t = \{w \in \mathbb{R}^{t+2} : \|w\| \leq 1 \text{ and } y_i w^T \ell_i \geq 0,\ 1 \leq i \leq t\} \tag{8}$$

where ℓ_i represents the i-th partial Cholesky factor. Then, the cut probabilities $p_{x_j,y}$ satisfy

$$p_{x_j,y} = \mathbb{P}\left(y\, w^T \ell_j \geq 0\right)$$

In conclusion, we have shown that the cut probabilities $p_{x_j,y}$ only depend on two quantities: the partial Cholesky factor ℓ_j and a random sample $w \sim Unif(W_t)$. In particular, our formulation has three main advantages compared to previous works:

- *Low Memory Footprint*: ALuMA [15] has to compute the entire $N \times N$ Cholesky decomposition matrix while our method only needs the first t columns. This truncated computation is quite similar to the Incomplete Cholesky Decomposition techniques [1, 35] but without a dedicated pivot-selection strategy. In fact, our method does not aim to compute a low-rank approximation of the kernel matrix K but only to provide the means for efficiently sampling the version space.
- *Efficient Sampling*: Unlike KQBC [12], the version space sample w is independent of the particular data point x, translating into a more efficient estimation of the cut probabilities $p_{x,y}$ over large sets of data points.
- *Optimizations*: The Cholesky property allows us to introduce a novel sampling optimization, called *rounding cache*, which significantly speeds up our sampling procedure. Refer to Section 4.3.1 for more details.

A more complete discussion on the above points will be left to Section 4.

3.4 Approximations for Large Data Sets

In some cases, the data set $\mathcal{X}$ can be so large it does not fit in memory, or working with $\mathcal{X}$ directly is too costly. In these cases, we propose the *subsampling* procedure below:

(1) First, a random sample $S \subset \mathcal{X}$ is obtained
(2) Then, the AL procedure is run over S , returning a classifier $h \in \mathcal{H}$ achieving low-error over S
(3) Finally, h will be used for labeling the entire $\mathcal{X}$

The success of the above procedure relies on appropriately choosing the sample size S. On one hand, choosing a small sample can greatly reduce the computational cost of the AL procedure. On the other hand, if S is too small, then the classifier h will probably not generalize well to the entire $\mathcal{X}$, especially for more complex hypothesis classes. This delicate balance in choosing the appropriate sample size is captured by the following lemma, which is a simple application of Theorem 2.2 from Mohri et al. [28]:

Definition 3.11. Let $\mathcal{X}$ be a data set, and let h^* be the target classifier. Then, we define the **accuracy error** of a classifier h over $S \subset \mathcal{X}$ to be

$$\epsilon_{acc}(h, S) = \frac{1}{|S|} \sum_{x \in S} \mathbb{1}(h(x) \neq h^*(x))$$

Lemma 3.12. *Let $\mathcal{X} = \{x_1, \ldots, x_N\}$ be a large data set and let $S = \{X_i\}_{i=1}^m$ be a sample with replacement from $\mathcal{X}$. Additionally, consider any hypothesis space $\mathcal{H}$ satisfying the realizability axiom,*

and let h^* be the target classifier. Then, for any $\delta \in (0,1)$, with probability at least $1-\delta$ we have

$$\forall h \in \mathcal{H},\ \epsilon_{acc}(h,\mathcal{X}) \leq \epsilon_{acc}(h,S) + \sqrt{\frac{\log|\hat{\mathcal{H}}| + \log\frac{2}{\delta}}{m}}$$

where $\hat{\mathcal{H}}$ is the set of equivalence classes defined in Section 3.1.

From the above lemma, irrespectively of which classifier $h \in \mathcal{H}$ the AL procedure computes, it is guaranteed to achieve low error over $\mathcal{X}$ under two conditions:

- *Enough data points are labeled by the user*: as more data points are labeled, the AL algorithm can compute an increasingly more accurate classifier h over S. In particular, choosing a fast converging AL algorithm lets us achieve low error rates with minimal labeling effort.
- *The sample size is large enough*: m should be chosen to control the complexity of our hypothesis class. In particular, for a given error rate η we need to sample at least

$$m = \frac{\log|\hat{\mathcal{H}}| + \log\frac{2}{\delta}}{\eta^2}$$

  data points. We also note that although $\hat{\mathcal{H}}$ is usually exponentially large, it can still lead to an acceptable sample size in many cases. For example, if $\mathcal{H}$ has finite VC-dimension D then by Sauer's Lemma [32] $|\hat{\mathcal{H}}| = O(N^D)$, resulting in a logarithmic dependency of m on the data set size.

In the interactive model development scenario, data sets often suffer from high class imbalance, especially in data exploration tasks. For such cases, it would be ideal to obtain generalization bounds for performance metrics other than accuracy, such as precision, recall, and F-score. This is explored in our next result:

Definition 3.13 (Precision, Recall, and F-score errors). For any two labels $r, s \in \{\pm 1\}$, let us define the confusion matrix entry by

$$C^{r,s}(h,S) = \frac{1}{|S|}\sum_{x\in S} \mathbb{1}(h(x) = r, h^*(x) = s)$$

With this, we define the following error quantities

$$\epsilon_{prec}(h,S) = 1 - precision(h,S) = \frac{C^{+,-}}{C^{+,+} + C^{+,-}}$$
$$\epsilon_{rec}(h,S) = 1 - recall(h,S) = \frac{C^{-,+}}{C^{+,+} + C^{-,+}}$$
$$\epsilon_{fscore}(h,S) = 1 - fscore(h,S) = \frac{C^{+,-} + C^{-,+}}{2C^{+,+} + C^{+,-} + C^{-,+}}$$

Theorem 3.14. *Let h be any classifier. Then, we have*

$$\epsilon_{fscore}(h,S) \leq \max(\epsilon_{prec}(h,S), \epsilon_{rec}(h,S)) \leq \frac{1}{\mu}\epsilon_{acc}(h,S)$$

where $\mu = \mathbb{P}(Y=1)$ is the positive class selectivity.

By combining the above result with Theorems 3.12, it is possible to achieve similar error guarantees in precision, recall, and F-score by having a sample $\frac{1}{\mu^2}$-times larger. However, whether this extra factor is acceptable in practice will depend on the use case. For example, data exploration tasks tend to be very selective, with the user interest usually covering $\mu = 1\%$ or less of the entire database, which corresponds to an increase in sample size by a factor of 10,000 or more. Refer to Appendix D for concrete examples of data exploration tasks and their selectivity factors.

3.4.1 The Majority Vote Classifier. In the particular case of version space algorithms, it is usually not specified how to choose a low-error classifier h. This is because the procedure is assumed to be run until a single hypothesis remains in the version space and, consequently, all labels are known. However, the lack of suitable convergence criteria and the high cost of manual labeling make this procedure impractical. In the literature, different methods were devised to overcome this limitation, such as: training an SVM classifier over the labeled set [37], choosing a random h in the version space [12], or, more recently, computing a *majority vote* of classifiers sampled from the version space [15]. In our work, we also adopt the majority voting idea:

Definition 3.15 (Majority Vote classifier). Let $\mathcal{X} = \{x_1, \ldots, x_N\}$ be a data set, and let $\mathcal{H}$ be a set of classifiers. Also, assume a probability distribution π over the set of equivalence classes $\hat{\mathcal{H}}$. The majority vote classifier is defined as

$$MV^{\pi}(x) = \arg\max_{y \in \mathcal{Y}} p^{\pi}_{x,y}$$

with $p^{\pi}_{x,y} = \mathbb{P}_{h\sim\pi}(h(x) = y)$.

The MV classifier has a very intuitive definition: for any data point, its predicted label is an agreement between the predictions of all classifiers in $\mathcal{H}$, weighted by π. In the particular case of VS algorithms, π can be chosen as the restriction of the prior probability to the version space, which makes $p^{\pi}_{x,y}$ coincide with the cut probability definition (2).

One advantage of the MV classifier is that it has interesting generalization properties:

Theorem 3.16. *For any $S \subset \mathcal{X}$ and any distribution π, the MV^{π} classifier satisfies*

$$\epsilon_{acc}(MV^{\pi}, S) \leq 2\mathbb{E}_{h\sim\pi}[\epsilon_{acc}(h, S)]$$

In particular, under the conditions of Theorem 3.12, with probability $1 - \delta$ it holds

$$\epsilon_{acc}(MV^{\pi}, \mathcal{X}) \leq 2\mathbb{E}_{h\sim\pi}[\epsilon_{acc}(h, S)] + \sqrt{\frac{2}{m}\left(\log|\hat{\mathcal{H}}| + \log\frac{2}{\delta}\right)}$$

From the above result, we can see that the MV classifier possesses similar error bounds as any $h \in \mathcal{H}$ but depends only on the *average training error* according to π. In the particular case of VS algorithms, π becomes more and more concentrated around the target classifier h^* as more points are labeled, which consequently reduces the average training error.

4 IMPLEMENTATION AND OPTIMIZATIONS

In this section, we detail how to efficiently implement the results of Theorem 3.10, resulting in an optimized version-space-based active learner called OptVS. At the core of our selection strategy and label prediction (majority voting) lies the computation of the cut probabilities $p_{x,y}$, which is done via Algorithm 1 below. In what follows, we give a detailed description of each step in this algorithm.

4.1 Computing the Partial Cholesky Factors ℓ_j

From Theorem 3.10, the first step in estimating $p_{x_j,+}$ is to compute the *partial Cholesky factor* ℓ_j, as defined in Equation (7). In particular, this vector can be easily computed once we have $\tilde{L}_j = (L^1_j, \ldots, L^t_j)$, where L is the Cholesky decomposition of the kernel matrix K. Now, from the equation $K = LL^T$ it follows that

$$LL_j = K_j \Rightarrow \tilde{L}\tilde{L}_j = \tilde{K}_j$$

Symbol	Description
d	Dimensionality of data points $x \in \mathbb{R}^d$
$\mathcal{E}(z, P)$	An ellipsoid in $\mathbb{R}^n$ centered at z and with scaling matrix P
B and T	Number of burn-in and thinning iterations in Markov Chain sampling
γ	Scaling factor for an ellipsoid: $\gamma\mathcal{E}(z, P) = \mathcal{E}(z, \gamma^2 P)$
$H(a, b)$	Half-space $\{x : a^T x \leq b\}$
t	Number of labeled examples at a given iteration of the AL process

Table 2. Notation table for Section 4

where $\tilde{L}$ is the upper-left $t \times t$ submatrix of L and $\tilde{K}_j$ is the t first components of K_j. With this, we note two things:

(1) $\tilde{L}$ corresponds to the Cholesky factor of $\tilde{K}$, the kernel matrix over the labeled data points (lines 1 and 2 in the pseudo-code).
(2) Since $\tilde{L}$ is lower-triangular, the system $\tilde{L}\tilde{L}_j = \tilde{K}_j$ can be efficiently solved via forward substitution (lines 8 and 9 in the pseudo-code).

Finally, assuming a kernel function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ of complexity $O(d)$ (which is the case for most popular kernel functions), computing the partial Cholesky factors incurs a complexity of $O\left(t^2(d+t) + |\mathcal{U}|t(d+t)\right)$. In terms of space complexity, this step is dominated by holding the matrices K and L in memory, which gives a bound of $O(t^2)$.

4.2 Estimating the Cut Probabilities via Sampling

Since the probabilities $p_{x_j,y}$ have no closed-form expression, we estimate these quantities via a *sampling procedure* [12, 15]. This step corresponds to lines 6 and 11 in Algorithm 1.

First, by the Law of Large Numbers, we can write

$$p_{x_j,y} \approx \frac{1}{M} \sum_{i=1}^{M} \mathbb{1}\left(y\, \ell_j^T w^i > 0\right)$$

where $\{w^i\}_{i=1,\ldots,M}$ is an i.i.d. sample following the uniform distribution over W_t (Eq. 8). Sampling uniformly over convex bodies like W_t is a well-known problem in convex geometry, for which we can apply the hit-and-run algorithm [25]. Hit-and-run generates a Markov Chain inside W_t converging to the uniform distribution; in other words, the longer the hit-and-run chain we generate, the closer to uniformly distributed our sample will be.

However, there is one issue with this sampling procedure: since we are assuming $\{w^i\}_{i=1}^M$ to be an independent sample, a separate hit-and-run chain must be generated for each w^i, which can become quite time-consuming in practice. To overcome this overhead, we opt for an optimized sampling procedure that requires generating a single chain. Since the hit-and-run chain $\{w^1, w^2, \ldots\}$ forms a positive Harris-recurrent Markov Chain [3], the Law of Large Numbers theorem for Markov Chains [20, Theorem 17.1.7] guarantees that

$$p_{x_j,y} = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\left(y\, \ell_j^T w^i > 0\right)$$

Thus, estimating $p_{x_j,y}$ can be done through a single chain. Furthermore, we employ two well-known techniques for improving the mixing time [17, 22]: *Burn-in* consists of discounting the first B samples from the chain since their distribution is very far from the stationary one; *Thinning*, on

Algorithm 1 Computing the cut probabilities

Input: labeled set $\mathcal{L} = \{(x_i, y_i)\}_{i=1}^t$, unlabeled set $\mathcal{U} \subset \{x \in \mathcal{X} : (x, \pm 1) \notin \mathcal{L}\}$, sample size M, any ellipsoid $\mathcal{E}_0 \supset W_t$, kernel function k

Output: The cut probabilities $p_{x_*,y}$, for $x_* \in \mathcal{U}$ and $y = \pm 1$

$K_{ij} \leftarrow k(x_i, x_j)$, $1 \le i, j \le t$
$L \leftarrow Cholesky(K)$
$\ell_i \leftarrow \left(1, L_i^1, \ldots, L_i^t, 0\right)$, $1 \le i \le t$
$W_t \leftarrow \{w \in \mathbb{R}^{t+2} : \|w\| \le 1 \text{ and } y_i \ell_i^T w \ge 0,\ 1 \le i \le t\}$
$\mathcal{E}_{round} \leftarrow rounding_algorithm(W_t, \mathcal{E}_0)$
$\{w^i\}_{i=1,\ldots,M} \leftarrow hit_and_run\,(W_t, M, \mathcal{E}_{round})$
for all $x_* \in \mathcal{U}$ **do**
  $K_* \leftarrow [k(x_1, x_*), \ldots, k(x_t, x_*)]^T$
  $L_* \leftarrow L^{-1} K_*$
  $\ell_* \leftarrow \left(1, L_*^1, \ldots, L_*^t, \sqrt{k(x_*, x_*) - \|L_*\|^2}\right)$
  $p_{x_*,y} \leftarrow \frac{1}{M} \sum_{i=1}^M \mathbb{1}\left(y\, \ell_*^T w^i > 0\right)$
end for
return $\{p_{x_*,y},\ \text{for } x_* \in \mathcal{U} \text{ and } y = \pm 1\}$

the other hand, only keeps one sample every T iterations after burn-in, in hopes to reduce sample correlation. In the end, we return $\{w^B, w^{B+T}, \ldots, w^{B+(M-1)T}\}$ as the final sample.

The technical details surrounding hit-and-run's implementation will be left in Appendix B. In particular, we note that the hit-and-run sampling has a time complexity of $O(Bt^2 + MTt^2)$, while the cut probability estimation takes $O(|\mathcal{U}|Mt)$. In terms of space, we have a complexity of $O(Mt)$ which is the size of the sample $\{w^i\}$ above returned by hit-and-run.

4.3 Improving the Sample Quality via Rounding

The hit-and-run algorithm can have a large mixing time, especially when the convex body $W \subset \mathbb{R}^n$ is very elongated in one direction. In practice, this can lead to poor sample quality, whose distribution is far from uniform even after thousands of steps; see Figure 1 for an example of this behavior.

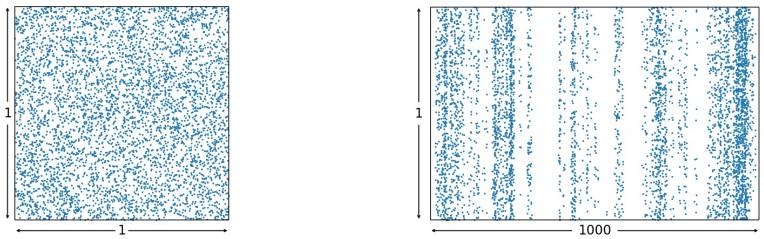
To address the above problem, we adopt a *rounding procedure* [8, 24]. In general terms, rounding is a preprocessing algorithm that attempts to make W more evenly elongated across all directions via a linear transformation $\mathcal{T}$. Once this transformation is found, the hit-and-run algorithm is run over $\mathcal{T}(W)$, and a sample over W is finally obtained via the inverse transformation $\mathcal{T}^{-1}$; see Figure 2(a) for an illustration of this process.

One well-known method for computing a rounding transformation is via Lovász's algorithm [24]. Let $\mathcal{E}(z, P) = \{w \in \mathbb{R}^n : (w-z)^T P^{-1} (w-z) \le 1\}$ denote an ellipsoid with *center* z and *scaling matrix* P.[5] The main idea is to find a *rounding ellipsoid* $\mathcal{E}$ satisfying

$$\gamma\mathcal{E} \subset W \subset \mathcal{E} \tag{9}$$

where $\gamma\mathcal{E} = \mathcal{E}(z, \gamma^2 P)$ is obtained by scaling the axes of $\mathcal{E}$ by a factor $\gamma > 0$. If γ is large enough, $\mathcal{E}$ should approximately capture the directions of major elongation of W; thus, by choosing $\mathcal{T}$ as any linear transformation mapping $\mathcal{E}$ into a unit ball, any major stretching of W should also be corrected, and the hit-and-run chain should mix quickly. More precisely, the resulting hit-and-run chain is guaranteed to mix in $O^*(n^2/\gamma^2)$ steps.

[5]The scaling matrix must be symmetric and positive-definite.



(a) A 1×1 square (b) A 1000×1 rectangle

Fig. 1. Distribution of the first 5000 samples generated by the hit-and-run procedure over a square and a long rectangle. As we can see, the samples distribution is closer to uniform for evenly elongated (or "round") convex bodies, thus implying a smaller mixing time.

All that remains now is how to compute a rounding ellipsoid. The main idea behind Lovász's algorithm is to construct an approximation of the *minimum volume ellipsoid* $\mathcal{E}^*$ containing W. The reason is that $\mathcal{E}^*$ is known to satisfy $\frac{1}{n}\mathcal{E}^* \subset W \subset \mathcal{E}^*$; thus, by computing an ellipsoid close enough to $\mathcal{E}^*$, one could hope to obtain a γ factor close to $1/n$ as well.

Algorithm 2 presents an outline of the ellipsoid computation algorithm, with a complete description deferred to Appendix C. Starting from any ellipsoid $\mathcal{E}_0 \supset W$, the algorithm constructs a sequence $\{\mathcal{E}_1, \mathcal{E}_2, \ldots\}$ that gets step by step closer to $\mathcal{E}^*$. More precisely, given $\mathcal{E}_k \supset W$, it proceeds as follows:

(1) First, the algorithm checks whether $\gamma\mathcal{E}_k \subset W$ for a given threshold $0 < \gamma < 1/n$. If so, $\mathcal{E}_k$ is returned as a rounding ellipsoid.
(2) Otherwise, it constructs a *separating half-space* $H(a, b) = \{x : a^T x \leq b\}$ satisfying $W \subset H(a, b)$ and $\alpha(\mathcal{E}_k, H) \leq -\gamma$, where

$$\alpha(\mathcal{E}, H) = \frac{a^T z - b}{\sqrt{a^T P a}}$$

See Figure 2(b) for an illustration.
(3) Finally, we set $\mathcal{E}_{k+1}$ as the *minimum volume ellipsoid* containing $\mathcal{E}_k \cap H(a, b)$, which can be analytically computed via Algorithm 6.

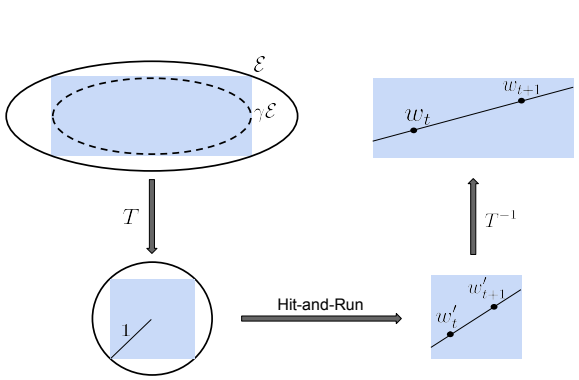
The above properties ensure that this algorithm will always terminate in a finite number of steps. This is because of two factors: first, $W \subset H$ implies that $W \subset \mathcal{E}_k$ for all k; second, the property $\alpha(\mathcal{E}_k, H) \leq -\gamma$ guarantees [4] that $vol(\mathcal{E}_{k+1}) < c \cdot vol(\mathcal{E}_k)$, where $0 < c < 1$ is a constant depending only on γ and n. When put together, these factors ensure an upper bound of

$$\frac{\log\left(vol(\mathcal{E}_0)/vol(W)\right)}{\log\left(1/c\right)} \tag{10}$$

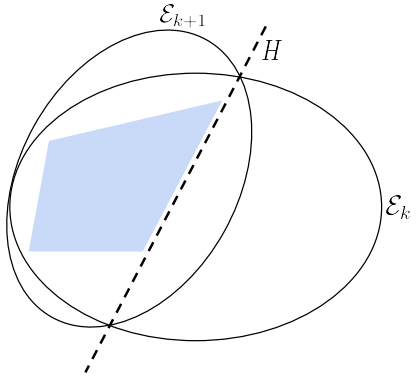
in the number of iterations until the rounding ellipsoid is found.

Despite the generality and guarantees of the rounding algorithm, it can be very costly to run in practice, especially as the dimensionality increases. This is especially problematic for our version space algorithm: every time we receive a new labeled point, not only does the dimensionality of W_t increase, but W_t itself is modified, and a new rounding ellipsoid must be computed. Thus, to improve its efficiency, we propose two major optimizations:

(1) *Ellipsoid caching* (Section 4.3.1): In practice, we note that the rounding algorithm can take a high number of iterations to converge. To address this issue, we introduce a novel *ellipsoid*



(a) A rounding transformation T corrects the elongation of the convex body (in blue) before generating the hit-and-run samples.



(b) Illustration of the rounding ellipsoid algorithm. A bounding ellipsoid $\mathcal{E}_k$ is repeatedly shrunk via hyperplane cuts H.

Fig. 2. Illustration of rounding procedure over a convex body (in blue).

Algorithm 2 Ellipsoid computation algorithm

Input: convex body $W \subset \mathbb{R}^n$, ellipsoid $\mathcal{E}_0 \supset W$, threshold $0 < \gamma < 1/n$
Output: An ellipsoid $\mathcal{E}$ satisfying $\gamma\mathcal{E} \subset W \subset \mathcal{E}$

```
k ← 0
while γ𝓔_k ⊄ W do
    H ← get_separating_halfspace(𝓔_k, W)
    𝓔_{k+1} ← minimum_volume_ellipsoid(𝓔_k, H)
    k ← k + 1
end while
return 𝓔_k
```

caching procedure that re-uses previous rounding ellipsoids $\mathcal{E}_0$ of small volume. In other words, $\mathcal{E}_0$ is used as a *warm-start* for the ellipsoid algorithm.

(2) *Optimized ellipsoid algorithm* (Section 4.3.2): The algorithm proposed by Lovász [24] was conceived for general convex bodies. However, its generality comes with a price: round-off errors introduced by every rounding iteration tend to accumulate, sometimes leading to numerical instability issues. We introduce a novel rounding algorithm that leverages the particular format of W_t and allows for a numerically stable implementation.

4.3.1 The Ellipsoid Caching Algorithm. To tackle the high number of rounding iterations, we introduce an *ellipsoid caching* procedure. Based on Equation (10), one way of reducing the number of iterations is to find an initial ellipsoid $\mathcal{E}_0$ of volume as small as possible to warm-start the rounding algorithm. With this idea in mind, our strategy is to construct $\mathcal{E}_0 \supset W_{t+1}$ by re-using the rounding ellipsoid $\mathcal{E}_{rnd} \supset W_t$ from a previous labeling iteration. Since $\mathcal{E}_{rnd}$ is an approximation of the minimum volume ellipsoid, $\mathcal{E}_{rnd}$ and W_t should also be close in size, which gives us hope in constructing $\mathcal{E}_0$ of volume close to W_{t+1} as well.

First, let us recall that $W_t = \{w \in \mathbb{R}^{t+2} : \|w\| \leq 1 \text{ and } y_i w^T \ell_i \geq 0,\ 1 \leq i \leq t\}$ as defined in Equation (8), t being the number of labeled points. When a new labeled point is obtained, the set W_t is subject to three modifications:

(1) The dimension increases by one
(2) Each ℓ_i is appended with a 0 to the right, for $1 \leq i \leq t$
(3) A new linear constraint $y_{t+1}\ell_{t+1}^T w \geq 0$ is included

To simplify our computation, we aim to find an ellipsoid $\mathcal{E}_0 \supset W_{t+1}$ that is independent of the new constraint ℓ_{t+1} added. In particular, the above properties guarantee it is enough to find $\mathcal{E}_0$ containing $W_t \times [-1, 1] \supset W_{t+1}$. With this insight, we can easily prove the following result:

Lemma 4.1. *Let $\mathcal{E}(z, P)$ be any ellipsoid containing W_t. Then, the ellipsoid $\mathcal{E}_0(z', P')$ given by*

$$z' = \begin{bmatrix} z \\ 0 \end{bmatrix} \qquad P' = (t+3)\begin{bmatrix} \frac{1}{t+2}P & 0 \\ 0 & 1 \end{bmatrix}$$

contains $W_t \times [-1, 1]$ and, thus, can be used as the starting ellipsoid for the rounding algorithm.

From the above lemma, we can note two main points. First, the ellipsoid $\mathcal{E}_0$ can be efficiently computed from the rounding ellipsoid $\mathcal{E}_{rnd}$, incurring a negligible performance penalty. Second, since the rounding ellipsoid $\mathcal{E}_{rnd}$ is very "tight" around W_t, then $\mathcal{E}_0$ should not be far from being a rounding ellipsoid for W_{t+1} as well, and the rounding procedure should intuitively converge in few iterations.

4.3.2 An Optimized Rounding Algorithm. In the original ellipsoid computation algorithm of Lovász [24], checking if $\gamma\mathcal{E} \subset W$ requires computing the extreme points of $\mathcal{E}$, a process that uses the eigenvalues and eigenvectors of the scaling matrix P. However, due to round-off errors introduced by each rounding iteration, P may no longer be positive-definite, making it infeasible to compute the extreme points. To avoid these numerical issues, we introduce a new ellipsoid computation algorithm for convex bodies of the form $W = \{w \in \mathbb{R}^n : \|w\| \leq 1 \text{ and } a_i^T w \leq 0,\ 1 \leq i \leq m\}$ which avoids the diagonalization step and allows for a numerically stable implementation.

In our version of the ellipsoid algorithm, we follow the same procedure outlined in Algorithm 2. In particular, this means that there are two main steps to consider: how to check if $\gamma\mathcal{E} \subset W$ and, if that is not the case, find a half-space H satisfying $\alpha(\mathcal{E}, H) > -\gamma$. To tackle the first problem, we rely on the following result:

Lemma 4.2. *Consider the convex body $W = \{w \in \mathbb{R}^n : \|w\| \leq 1 \text{ and } a_i^T w \leq 0,\ 1 \leq i \leq m\}$ and let $\mathcal{E}(z, P)$ be an ellipsoid. Then, for any $\gamma > 0$ we have*

$$\gamma\mathcal{E} \subset W \iff \max_{1 \leq i \leq m} \alpha_i \leq -\gamma \text{ and } \max_{\|w\|=1} \alpha_w \leq -\gamma$$

where $\alpha_i = \alpha(\mathcal{E}, H(a_i, 0)) = z^T a_i / \sqrt{a_i^T P a_i}$ and $\alpha_w = \alpha(\mathcal{E}, H(w, 1)) = (z^T w - 1)/\sqrt{w^T P w}$.

The above result also helps with the solution for the second part of the algorithm, that is, finding a cutting half-space $H(a, b)$ satisfying $\alpha(\mathcal{E}, H) > -\gamma$. First, let's define $i^* = \arg\max_i \alpha_i$ and $w^* = \arg\max_{\|w\|=1} \alpha_w$. If $\gamma\mathcal{E} \not\subset W$ then either $\alpha_{i^*} > -\gamma$ or $\alpha_{w^*} > -\gamma$, meaning that either $H(a_{i^*}, 0)$ or $H(w^*, 1)$ can be chosen as cutting half-space.

Now, the only point remaining is how to compute $\max_{\|w\|=1} \alpha_w$. This is a non-linear, constrained optimization problem for which many efficient solvers exist (e.g. SLSQP, COBYLA). However, since calling the solver can be expensive if done repeatedly, we introduce two small optimizations:

- Only call the solver if $\alpha_{i^*} \leq -\gamma$: in other words, we use the cuts $H(a_i, 0)$, which are relatively cheap to compute, until no longer possible.

Algorithm 3 Optimized rounding algorithm

Input: convex body $W = \{w \in \mathbb{R}^n : \|w\| \leq 1 \wedge a_i^T w \leq 0,\ 1 \leq i \leq m\}$, ellipsoid $\mathcal{E}_0 = \mathcal{E}(z_0, P_0) \supset W$, threshold $0 < \gamma < \frac{1}{n}$
Output: An ellipsoid $\mathcal{E}$ satisfying $\gamma\mathcal{E} \subset W \subset \mathcal{E}$

$k \leftarrow 0$
while True **do**
  $\alpha_{i^*} \leftarrow \max_i \frac{z_k^T a_i}{\sqrt{a_i^T P_k a_i}}$
  $\alpha_z \leftarrow \frac{\|z_k\|(\|z_k\|-1)}{\sqrt{z_k^T P_k z_k}}$
  if $\alpha_{i^*} \geq \alpha_z$ and $\alpha_{i^*} > -\gamma$ **then**
    $\mathcal{E}_{k+1} \leftarrow minimum_volume_ellipsoid(\mathcal{E}, H(a_{i^*}, 0))$
  else if $\alpha_z \geq \alpha_{i^*}$ and $\alpha_z > -\gamma$ **then**
    $\mathcal{E}_{k+1} \leftarrow minimum_volume_ellipsoid(\mathcal{E}, H(z_k, \|z_k\|))$
  else
    $\alpha_{w^*} \leftarrow \max_{\|w\|=1} \frac{w^T z_k - 1}{\sqrt{w^T P_k w}}$
    if $\alpha_{w^*} \leq -\gamma$ **then**
      return $\mathcal{E}_k$
    end if
    $\mathcal{E}_{k+1} \leftarrow minimum_volume_ellipsoid(\mathcal{E}_k, H(w^*, 1))$
  end if
  $k \leftarrow k + 1$
end while

- Only call the solver if $\alpha_z = \alpha(\mathcal{E}, H(z, \|z\|)) \leq -\gamma$: the cut $H(z, \|z\|)$ has the interesting property that $\alpha_z < 0 \iff \|z\| < 1$. In particular, the condition $\alpha_z \leq -\gamma < 0$ allows us to avoid calling the solver when $z \notin W$.

Finally, by putting together all of the above considerations, we are now ready to present our optimized rounding algorithm for W, described in Algorithm 3.

One last point to discuss is the numerical stability of our rounding algorithm. A main point of concern is computing the quantity $1/\sqrt{a^T P a}$ for some vector $a \neq 0$. As we previously discussed, round-off errors may cause P_k to no longer be positive-definite, which leads to $a^T P a \leq 0$. Similarly, this problem is also known to occur within the minimum volume ellipsoid routine, for which a few solutions have already been proposed in the literature [4, 13].

Our method of choice is to use the LDL^T decomposition of P [13]. By doing so, we can write $a^T P a = \sum_i D_{ii}(L_i^T a)^2$, which is guaranteed to be positive. Additionally, the minimum volume ellipsoid routine can be modified to directly update the matrices L and D in a numerically stable way, and we no longer have to store the scaling matrix P; refer to Appendix C for the complete implementation details. We also note that although this procedure could also be applied to Lovász's rounding algorithm, we would still have to store or compute the scaling matrix P since it is necessary for the diagonalization step. In contrast, our solution removes the diagonalization step and eliminates the dependency on P via the decomposition, thus solving the numerical instability issue.

4.4 Hit-and-run's Starting Point

Hit-and-run starts by finding a point w^0 inside W_t. Although this could be done by solving a linear programming task, it can take a significant amount of time. Instead, we rely on the rounding procedure: since the computed ellipsoid $\mathcal{E}(z, P)$ satisfies $\gamma\mathcal{E} \subset W_t$ for some $\gamma > 0$, it also guarantees that $z \in W_t$, which can be used as a starting point for hit-and-run.

5 A FACTORIZED VERSION SPACE ALGORITHM

To improve the efficiency of version space (VS) algorithms on large data sets, we aim to augment them with additional insights obtained in the user labeling process. In particular, we observe that when a user labels a data instance, the decision-making process can often be broken into a set of simple, independent "yes" or "no" questions that, when combined, derive the final answer. Let's consider the following example: when a customer decides whether a car model is of interest, she can have the following questions in mind:

- Q^1: Is the gas mileage good enough?
- Q^2: Is the vehicle spacious enough?
- Q^3: Is the color a preferred one?

We do not expect the user to specify his questions precisely as classification models, which would require knowing the exact shape of the decision function and all constants used. In fact, we only expect the user to have a high-level intuition of the set of questions and their related attributes.

Factorization Structure. Formally, we model the user intuition of the set of questions and their relevant attributes as a factorization structure. Let us model the decision-making process using a complex question Q defined on an attribute set $\boldsymbol{A}$ of size d. Based on user intuition, Q can be broken into simple independent questions $\{Q^1, \ldots, Q^S\}$ with each Q^s posed on a subset of attributes $\boldsymbol{A}^s \subset \boldsymbol{A}$. The family of attribute sets $(\boldsymbol{A}^1, \cdots, \boldsymbol{A}^S)$ may be disjoint or overlapping in attributes as long as the user believes that decisions for these smaller questions can be made independently. $(\boldsymbol{A}^1, \cdots, \boldsymbol{A}^S)$ is referred to as the *factorization structure*.

The independence assumption in the decision process should not be confused with the data correlation issue. For example, the color and size of cars can be statistically correlated, e.g., large cars are often black. But the user decision does not have to follow the data characteristics; e.g., the user may be interested in large cars that are red. As long as the user believes that his decisions for the color and size are independent, the factorization structure ({color}, {size}) applies. On the contrary, if the two attributes are not independent in the decision-making process, e.g., the user prefers red for small cars and black for large ones, but the year of production is an independent concern, then the factorization structure can be ({color size}, {year}).

Decision functions. Given a data instance x, we denote $x^s = proj(x, \boldsymbol{A}^s)$ the projection of x over attributes $\boldsymbol{A}^s$. We use $Q^s(x^s) \rightarrow \{+, -\}$ to denote the user's decision on x^s. Then, we assume that the final decision from the answers to these questions is a boolean function $F : \{+, -\}^S \rightarrow \{+, -\}$:

$$Q(x) = F(Q^1(x^1), \ldots, Q^S(x^S)) \tag{11}$$

The most common example of a decision function is the conjunctive form, meaning that the user requires each small question to be + to label the overall instance with +. Given that any Boolean expression can be written in the conjunctive normal form, $Q^1(x^1) \wedge \ldots \wedge Q^S(x^S)$ already covers a large class of decision problems, and it will be considered the default choice. We note that our work can also support any other choice of decision function, but then it must be specified by the user.

Given the decision function F, we aim to learn the subspatial decision functions, $\{Q^1(x^1), \ldots, Q^S(x^S)\}$, efficiently from a small set of labeled instances. For a labeled instance x, the user provides a collection of *subspatial labels* $(y^1, \ldots, y^S) \in \{+, -\}^S$ to enable learning. We note that,

Symbol	Description
S	Number of subspaces in a factorization structure
Q and Q^s	Query representing the user interest and its decomposition into simpler sub-queries
F	Decision function combining all Q^s into Q: $Q(x) = F(Q^1(x^1), \ldots, Q^S(x^S))$
$(\boldsymbol{A}^1, \ldots, \boldsymbol{A}^S)$	Factorization structure, $\boldsymbol{A}^s$ being the set of attributes involved in sub-query Q^s
$\mathcal{H}_f$	Factorized hypothesis space $\mathcal{H}^1 \times \ldots \times \mathcal{H}^S$
$\mathcal{H}^s$	Set of classifiers modeling sub-query Q^s
H	Factorized classifier $(h^1, \ldots, h^S) \in \mathcal{H}_f$ with $h^s \in \mathcal{H}^s$
$\mathcal{V}_f$	Factorized version space $\mathcal{V}^1 \times \ldots \times \mathcal{V}^S$ with $\mathcal{V}^s$ being the VS over subspace s
π_f	Prior over $\mathcal{H}_f / \sim$; under independence assumption can be factorized as $\pi^1 \times \ldots \times \pi^S$
$p^s_{x,\pm}$	Cut probability for each positive / negative class in subspace s
C	Set of categories in a categorical subspace
T^s, S^s	Time and space complexities in computing $p^s_{x,\pm}$

Table 3. Notation table for Section 5

while this process requires more annotation effort from the user, it should not demand more *thinking* effort than deriving the global label.

Generality. In what follows, we compare our factorization framework with similar works in the literature. First, we note the differences in our factorization framework from Huang et al. [18]:

- Our factorization is applied to version space algorithms, while Huang et al. [18] does not consider them at all.
- We eliminate the assumption that either the set $\{x^s : Q^s(x^s) = +\}$ or the set $\{x^s : Q^s(x^s) = -\}$ are convex objects.
- The global decision function F must be conjunctive in Huang et al. [18], which is relaxed to any boolean expression in our work. More precisely, our factorized algorithms can support any non-conjunctive decision function, but in those cases, the user should provide its particular expression at the beginning of the exploration process.

Another line of related work is the data programming systems such as Snorkel [2, 31]. In such systems, users have to provide labeling functions whose predictions are correlated with the labeling task at hand. However, the user usually has to manually tune one or more parameters in such functions, which can have a large impact on the prediction quality if not properly done. In more recent work, Snuba [39], such constants could be automatically tuned by relying on a small pool of hundreds to thousands of labeled points. In contrast, our factorization framework only requires knowing which sets of attributes compose each subspace; no initial pool of labeled data or manual tuning of parameters is required.

Finally, we also discuss how our idea of "factorization" compares with other works in Machine Learning. For example, we can mention the "matrix factorization" technique in recommender systems and the "factorization of probability distributions" in probabilistic graphical models. However, the general concept of "breaking something complex into several simple parts" remains in all of these works, including ours: in our case, the user prediction function $Q(x)$ is broken down into several simpler predictors $Q^s(x^s)$, which can then be pieced together into the final prediction $Q(x) = F(Q^1(x^1), \ldots, Q^S(x^S))$.

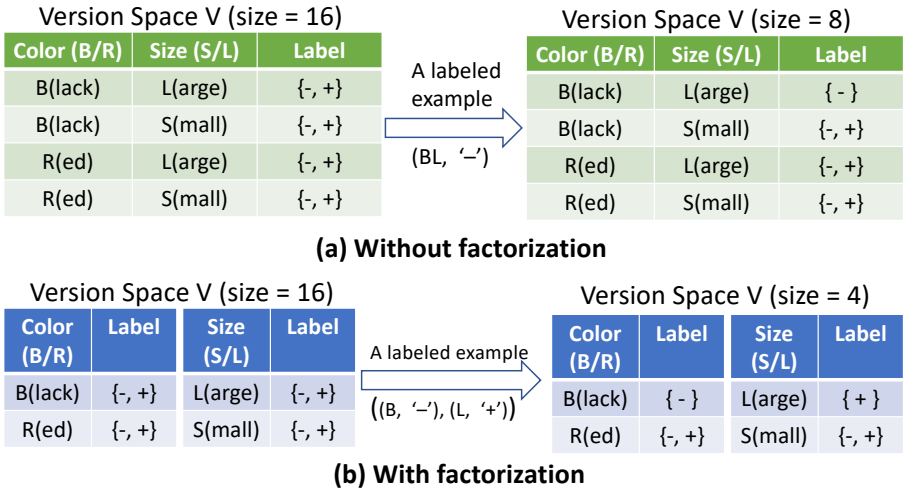


Fig. 3. Illustration of factorization on the version space.

5.1 Introduction to Factorized Version Space

We now give an intuitive description of the factorized version space, deferring a formal definition to Section 5.3.

Without factorization, our problem is to learn a classifier h (e.g., a kernel classifier) from a labeled data set $\mathcal{L} = \{(x_i, y_i)\}$, where x_i is a data instance on the attribute set $\boldsymbol{A}$ and $y_i \in \{+, -\}$. The version space $\mathcal{V}$ includes all possible configurations of h (e.g., all possible bias and weight vector of the kernel classifier) consistent with $\mathcal{L}$.

Given a factorization structure $(\boldsymbol{A}^1, \cdots, \boldsymbol{A}^S)$, we define S subspaces with the s-th subspace including all the data instances projected on $\boldsymbol{A}^s$. Then, our goal is to learn a classifier h^s for each subspace s from its labeled set $\mathcal{L}^s = \{(x_i^s, y_i^s)\}$, y_i^s being the subspatial label for x_i^s. For the classifier h^s, its version space, $\mathcal{V}^s$, includes all possible configurations of h^s that are consistent with $\mathcal{L}^s$. Across all subspaces, we can reconstruct the version space via $\mathcal{V}_f = \mathcal{V}^1 \times \ldots \times \mathcal{V}^S$. For any unlabeled instance, x, we can use $F(h^1(x^1), \ldots, h^S(x^S))$ to predict a label.

At this point, one may wonder what benefit factorization provides in the learning process. We use the following example to show that factorization may enable a faster reduction of the version space, hence enabling faster convergence to the correct classification model.

Example. Figure 3 shows an example in that the user considers the color and the size of cars, where the color can be black (B) or red (R), and the size can be large (L) or small (S), resulting in four combinations of vehicles in total. Figure 3(a) shows that without factorization and in the absence of any user-labeled data, the version space contains 16 possible classifiers that correspond to all possible combinations of $\{+, -\}$ labels assigned to the four types of cars. Once we obtain the '–' label for the type BL (color = Black and size = Large), the version space is reduced to the 8 classifiers that assign the '–' label to BL.

Next, consider factorization. In the absence of labeled data, Figure 3(b) shows that the subspace for color includes two types of cars, B and R, and its version space includes 4 possible classifiers that correspond to all combinations of $\{+, -\}$ labels over these two types of cars. Similarly, the subspace for size also includes 4 classifiers and, by combining both, we have 16 possible classifiers in total. Once BL is labeled, this time with two subspatial labels ((B,–), (L,+)), each subspace has only two classifiers left, yielding 4 remaining classifiers across the two subspaces. As can be seen,

Algorithm 4 A Factorized Version Space Algorithm

Input: data set $\mathcal{X}$, initial labeled set $\mathcal{L}_0$, data sample size m_1, per labeling iteration subsample size m_2, version space sample size M

1: $S \leftarrow subsample(\mathcal{X}, m_1)$
2: $\mathcal{L} = \mathcal{L}_0$, $\mathcal{U} = S$
3: **while** user is still willing **do**
4: $\quad \mathcal{U}' \leftarrow \text{subsample}(\mathcal{U}, m_2)$
5: $\quad$ **for** each subspace s **do**
6: $\quad\quad \mathcal{U}^s \leftarrow \{x^s, \text{ for } x \in \mathcal{U}'\}$
7: $\quad\quad \mathcal{L}^s \leftarrow \{(x^s, y^s), \text{ for } (x, y) \in \mathcal{L}\}$
8: $\quad\quad \{p^s_{x,+}(x)\}_{x \in \mathcal{U}'} \leftarrow \text{compute_cut_probability}(\mathcal{L}^s, \mathcal{U}^s, M)$
9: $\quad$ **end for**
10: $\quad x^* \leftarrow \text{selection_strategy}(\mathcal{U}', p^1_{x,+}, \ldots, p^S_{x,+})$
11: $\quad y^* \leftarrow \text{get_labels_from_user}(x^*)$
12: $\quad \mathcal{L}, \mathcal{U} \leftarrow \mathcal{L} \cup \{(x^*, y^*)\}, \mathcal{U}/\{x^*\}$
13: **end while**
14: **for** k from 1 to S **do**
15: $\quad MV^s \leftarrow \text{train_majority_vote_classifier}(\mathcal{L}^s)$
16: **end for**
17: $h_{final} \leftarrow x \mapsto F(MV^1(x^1), \ldots, MV^S(x^S))$
18: **return** h_{final}

with factorization, each labeled instance offers more information and hence can lead to a faster reduction of the version space.

5.2 Overview of a Factorized Version Space Algorithm

Based on the above insight, we propose a new active learning algorithm called the *factorized version space algorithm*. It leverages the factorization structure provided by the user to create subspaces and factorizes the version space accordingly to perform active learning in each subspace.

Algorithm 4 shows the pseudocode of our algorithm. It starts by taking a labeled data set (which can be empty) and creating a memory-resident sample from the underlying large data set as an unlabeled pool $\mathcal{U}$ (lines 1-2). This is done for efficiency reasons to guarantee the interactive performance of our algorithm, as discussed in Section 3.4. Then, it proceeds to an iterative procedure (lines 3-13): In each iteration, the unlabeled pool may be further subsampled to expedite learning, thus obtaining $\mathcal{U}'$ (line 4). Then the algorithm considers each subspace (lines 5-9), including both labeled instances, $(x^s, y^s) \in \mathcal{L}^s$, and unlabeled instances, $x^s \in \mathcal{U}^s$, projected to this subspace. The key step is to compute, for each unlabeled instance x, how much its projection x^s can reduce the version space $\mathcal{V}^s$ once its label is acquired (line 8). This step requires efficient sampling of the version space $\mathcal{V}^s$, as shown in Section 4. Once the above computation completes for all subspaces, the algorithm chooses the next unlabeled instance that can result in the best reduction of the factorized version space $\mathcal{V}_f = \mathcal{V}^1 \times \ldots \times \mathcal{V}^S$ (line 10). Our work proposes different strategies and proves the optimality of these strategies, as we detail in Sections 5.3–5.5. The selected instance is then presented to the user for labeling, and the unlabeled pool is updated. The algorithm then proceeds to the next iteration.

Once the user wishes to stop exploring, a majority vote classifier (Section 3.4.1) is trained for each subspace k (lines 14-16). Finally, given MV^s for each subspace, the algorithm builds the final

classifier $F(MV^1, \ldots, MV^S)$ (line 17), which can then be used to label all points in the entire data set $\mathcal{X}$.

5.3 Bisection Rule over Factorized Version Space

Let's suppose that a factorization structure $(\boldsymbol{A}^1, \cdots, \boldsymbol{A}^S)$ is given. For each subspace s, the user labels the projection x^s of x over $\boldsymbol{A}^s$ based on a hypothesis from a hypothesis class $\mathcal{H}^s$ with prior probability distribution π^s. The user then provides a binary label $\{+, -\}$ for each subspace; in other words, for each x a collection of *subspatial labels* $(y^1, \ldots, y^S) \in \mathcal{Y}_f = \{+, -\}^S$ is provided by the user.

Definition 5.1 (Factorized hypothesis and factorized hypothesis space). Define a factorized hypothesis as any function $H : \mathcal{X} \to \mathcal{Y}_f$ such that

$$H(x) = (h^1, \ldots, h^S)(x) = (h^1(x^1), \ldots, h^S(x^S))$$

H belongs to the product space $\mathcal{H}_f = \mathcal{H}^1 \times \ldots \times \mathcal{H}^S$, which we call the factorized hypothesis space.

We assume that the user labels the subspaces independently and consistently within each subspace.

Definition 5.2 (Factorized version space). Let $\mathcal{L}$ be the labeled set at any iteration of active learning. We define the factorized version space as

$$\mathcal{V}_f = \{H \in \hat{\mathcal{H}}_f : H(x) = y \text{ for all } (x, y) \in \mathcal{L}\}$$

Additionally, we note that $\mathcal{V}_f$ is equivalent to $\prod_s \mathcal{V}^s = \mathcal{V}^1 \times \ldots \times \mathcal{V}^S$, where

$$\mathcal{V}^s = \{h \in \hat{\mathcal{H}}^s : h(x^s) = y^s \text{ for all } (x, y) \in \mathcal{L}\}$$

is the version space at subspace s.

Given the assumption of independent labeling among the subspaces, the prior probability distribution over the factorized hypothesis space is $\pi_f = \pi^1 \times \ldots \times \pi^S$. Now, by applying the bisection rule to $(\mathcal{X}, \mathcal{Y}_f, \mathcal{H}_f, \pi_f)$, we can define we strategy below:

Definition 5.3 (Factorized greedy selection strategy). Let $p_{x,y} = \mathbb{P}(H(x) = y \,|\, H \in \mathcal{V}_f)$. The factorized greedy strategy is defined as

$$\arg\max_{x \in \mathcal{U}} 1 - \sum_{y \in \mathcal{Y}_f} p_{x,y}^2 \tag{12}$$

Additionally, by noting that $\min_{H \in \tilde{\mathcal{H}}} \tilde{\pi}(H) = \prod_s \min_{h^s \in \mathcal{H}^s} \pi^s(h^s)$, the following theorem is a direct application of Equation (3):

Theorem 5.4 (Number of iterations with factorization). *The factorized greedy strategy in* (12) *takes at most*

$$OPT_f \cdot \left(1 + \sum_{s=1}^{S} \log\left(\frac{1}{\min_{h^s} \pi^s(h^s)}\right)\right)^2$$

iterations in expectation to identify a hypothesis randomly drawn from π_f, where OPT_f is the minimum number of iterations across all strategies that continue until the target hypothesis over all subspaces has been found.

It is interesting to notice that OPT_f requires the classifiers to continue exploring until *the user's concept is found in every subspace*. As one can intuitively expect, the value of OPT_f is governed by the most complex subspace in the factorization in terms of data distribution and complexity of the hypothesis class $\mathcal{H}^s$.

Finally, we derive a simplified computation of the factorized greedy strategy (12):

Theorem 5.5. *Let $p^s_{x,\pm} = \mathbb{P}(h(x^s) = \pm \mid h \in \mathcal{V}^s)$. The factorized greedy selection strategy (12) is equivalent to*

$$\arg\max_{x \in \mathcal{U}} 1 - \prod_{s=1}^{S} (1 - 2\, p^s_{x,+} p^s_{x,-}) \tag{13}$$

5.4 Factorized Squared-Loss Strategy

We also propose a variant of the greedy strategy, called the *squared-loss strategy*, for selecting the next example for labeling:

$$\arg\max_{x \in \mathcal{U}} \sum_{s=1}^{S} 2 p^s_{x,+} p^s_{x,-} \tag{14}$$

Intuitively, this strategy follows the same intuition as the greedy strategy: find an example x that simultaneously halves all $\mathcal{V}^s$. This strategy also has very similar performance guarantees to the greedy strategy:

Theorem 5.6 (Factorized squared-loss strategy). *The squared-loss strategy takes at most*

$$S \cdot OPT_f \cdot \left(1 + \sum_{s=1}^{S} \log\left(\frac{1}{\min_{h^s} \pi^s(h^s)}\right)\right)^2$$

iterations in expectation.

5.5 Factorized Product-Loss Strategy

One problem with the previous strategies is that they attempt to find $H^* \in \mathcal{H}_f$ that correctly infers all subspatial labels, without taking into account the *final prediction* $F(H(x))$ of our classifier. For example, if changing a single subspatial label does not affect the final prediction $F(H(x))$, it is not worth spending any effort in reducing the version space of this particular subspace.

With this idea in mind, let us define a function $\tilde{F}$ mapping each $H \in \mathcal{H}_f$ into the classifier making the final predictions: $\tilde{F}(H)(x) = F(H(x))$. Given some labeled data $\mathcal{L}$, the set of final, consistent classifiers is given by

$$\tilde{\mathcal{V}}_f = \{\tilde{F}(H) : H(x) = y, \forall (x, y) \in \mathcal{L}\}$$

Now, our selection strategy follows the same principle of the version space bisection rule: select the point x which halves the set of consistent classifiers $\tilde{\mathcal{V}}_f$ in half, that is

$$\arg\max_{x \in \mathcal{U}} 2 \tilde{p}_{x,+} \tilde{p}_{x,-} \tag{15}$$

where $\tilde{p}_{x,\pm} = \mathbb{P}(F(H(x)) = \pm \mid [H] \in \mathcal{V}_f) = \sum_{y:F(y)=\pm} \prod_{k=1}^{S} p^s_{x,y_s}$. We call it the *product-loss* strategy because, in the popular case of conjunctive F, it simplifies to

$$\tilde{p}_{x,+} = \prod_{s=1}^{S} p^s_{x,+}$$

The above strategy is a hybrid between the unfactorized and factorized version space bisection rules. On one hand, it splits the version space according to the binary prediction $F(H(x)) \in \{-, +\}$, repeating this procedure until one single equivalence class $[\tilde{F}(H^*)]$ of binary classifiers remains. On the other hand, the version space $\tilde{\mathcal{V}}_f$ itself is composed of classifiers consistent with all subspatial labels and not only the final labels. In conclusion, the hybrid nature of this strategy should intuitively guarantee a faster convergence speed than our previous factorized strategies.

5.6 Optimization for Categorical Subspaces

The factorization framework also leads to a simple treatment of categorical variables. Assume that all attributes are categorical in a given subspace s, and let $C = \{x^s : x \in \mathcal{X}\}$ be the set of all different categories in the data. Instead of encoding these values to numbers (e.g., using the one-hot encoding), we define a categorical hypothesis class $\mathcal{H}^{cat} = \{h : C \to \{+, -\}\}$ to model the user interest over this subspace; in other words, each classifier represents a $\{+, -\}$-vector over categories indicating which are interesting or not. In particular, it is easy to implement the version space bisection rule over $\mathcal{H}^{cat}$: given some labeled data $\mathcal{L}^s$ and by assuming a uniform prior, the version space cut probabilities are easily shown to be

$$p_{c,+} = \begin{cases} 1, \text{ if } (c, +) \in \mathcal{L}^s \\ 0, \text{ if } (c, -) \in \mathcal{L}^s \\ 0.5, \text{ otherwise} \end{cases} \tag{16}$$

which can be directly plugged into Equations (13) or (14). During prediction, we simply use the majority vote classifier definition $MV^{cat}(c) = + \iff p_{c,+} > 0.5 \iff (c, +) \in \mathcal{L}$.

We also note that these cut probabilities are extremely efficient to compute: compared to our version space algorithm, it does not require any sampling procedure or optimization: a simple memorization of the categories and labels is enough.

5.7 Time and Space Complexity

Finally, we discuss the time and space complexity of the factorized algorithms in Sections 5.4–5.6.

Let's first look at time complexity. Let T^s be the time complexity of the algorithm estimating $p^s_{x,\pm}$ over all unlabeled examples $x \in \mathcal{U}$ in subspace s. Then, a naive (e.g., serial) implementation of our factorized strategies will take $O\left(\sum_{s=1}^{S} T^s + |\mathcal{U}|S\right)$ to complete, where the last term is the time taken to compute Equations 13–15. However, we can easily improve this complexity by computing all $p^s_{x,\pm}$ in parallel, which gives a complexity of $O\left(\max_s T^s + |\mathcal{U}|S\right)$.

Now, let's consider the space complexity. For each subspace s, we need to maintain some data structures in memory allowing us to efficiently compute $p^s_{x,\pm}$ (e.g. hit-and-run sample, rounding ellipsoid). If we denote such space complexity by S^s, then it is clear that a factorized strategy costs $\sum_{s=1}^{S} S^s$ of space, irrespective of the serial or parallel computation strategies discussed above.

Finally, we note that our results in Section 5.6 make categorical subspaces very lightweight to process, which can significantly reduce the final complexity of factorized strategies. In fact, each such subspace only costs $O(1)$ time and $O(|C|)$ space, where $|C|$ is the number of categories.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate our proposed techniques against state-of-the-art active learners [15, 18, 33, 37] in terms of accuracy (using F-score) and efficiency (execution time in each labeling iteration). Note that F-score is the harmonic mean of precision and recall for retrieving objects in the positive class and a suitable measure when the user interest covers a small portion of the data set, which is

Query	*sel* (%)	Dim	S	Is Convex
SDSS 01	0.1	2	1	T
SDSS 02	0.1	2	1	T
SDSS 03	0.1	2	1	T
SDSS 04	0.1	2	1	T
SDSS 05	0.01	4	2	T
SDSS 06	0.01	6	3	F
SDSS 07	7.8	4	2	F
SDSS 08	5.5	7	5	T
SDSS 09	1.5	5	4	T
SDSS 10	0.5	5	5	T
SDSS 11	0.1	5	5	F

Query	*sel* (%)	Dim	S	Is Convex
Car 01	0.32	6 (58)	6	T
Car 02	0.27	8 (37)	8	T
Car 03	0.27	8 (35)	6	T
Car 04	0.25	6 (14)	6	T
Car 05	0.23	6 (418)	6	T
Car 06	0.34	4 (49)	4	T
Car 07	0.82	6 (59)	6	T
Car 08	0.32	8 (26)	8	T
Car 09	0.30	4 (12)	4	T
Car 10	0.36	8 (31)	8	T
Car 11	0.29	6 (26)	6	T
Car 12	0.20	4	4	T
Car 13	0.29	10 (24)	10	T
Car 14	0.23	5 (29)	5	T
Car 15	0.91	6 (51)	6	T
Car 16	0.21	4 (12)	4	T
Car 17	0.20	6 (49)	6	T
Car 18	0.20	6 (17)	6	T

Table 4. Collection of the most relevant properties of each user interest pattern over SDSS and Car data sets, where *sel* is the selectivity, "Dim" the number of features, and S is the number of subspaces. If a query involves categorical attributes, the "Dim" column also displays in parenthesis the number of features *after* one-hot encoding.

often the case in large data set exploration. Our evaluation particularly focuses on F-score in the first 200 labeling iterations as our work aims to address the slow start problem. All algorithms are part of our Python-based prototype.[6]

6.1 Experimental Setup

Data sets and user interest patterns: We use two real-world data sets and user interest patterns to evaluate our techniques:

(1) *Sloan Digital Sky Survey* (SDSS, 190 million points): The data set contains the "PhotoObjAll" table with 510 attributes and 190 million sky observations.[7] We used a 1% sample (1.9 million points, 4.9GB) to create a data set for running active learning algorithms—our formal results in Section 3.4 allowed us to use a sample for efficient execution, yet being able to approximate the accuracy over the entire data set.
SDSS also offers a query release where the SQL queries reflect the data interest of scientists.[8] We extracted 11 queries to build a benchmark, as shown in Appendix D, and treated them as the ground truth of the positive classes of 11 classifiers to be learned—our system does *not* need to know these queries in advance but can learn them via active learning.
(2) *Car data set* (5622 points): This small data set was used in Huang et al. [18] to conduct a user study, which generated 18 queries representing the true user interests. Categorical attributes were preprocessed using one-hot encoding.

Table 4 contains a complete description of all user interest patterns. We have chosen these particular data sets because their user interest patterns are encoded as SQL queries, which enables the extraction of factorization structures and, consequently, the evaluation of our factorized techniques. The SQL text underlying each user pattern can be found in Appendix D.

[6]Our code is available at https://github.com/AIDEmeProject/PythonMiddleware
[7]http://www.sdss3.org/dr8/
[8]http://skyserver.sdss.org/dr8/en/help/docs/realquery.asp

Algorithms: We compare our algorithms to state-of-the-art VS strategies, including Simple Margin (SM) [37], Query-by-Disagreement (QBD) [33], ALuMA [15], as well as a factorization-aware algorithm, DSM [18]. When selecting the next point to label, each algorithm only considers a sample of at most 50,000 unlabeled points. In our experiments, each active learning algorithm starts with one positive and negative example chosen at random and runs up to 200 additional labeled examples (iterations). At every iteration, the classification accuracy of each active learner is computed over the entire data set used for training, which corresponds to the user's interest metric for practical use cases. Each experiment was repeated 10 times and individual results were averaged.

Hyper-parameter tuning: Hyper-parameters were tuned via a standard grid-search procedure to achieve the highest accuracy while running under interactive performance (or close to it). Based on the results of this procedure, we set hyper-parameters of different algorithms as follows: All active learners use an RBF kernel with a default scaling of 1/num_features, except for the SDSS 04 for which we used 0.001. Algorithms relying on SVM (SM, QBD, DSM) use a penalty of $C = 10^5$. For QBD, we use a background sample of 200 points weighted by 10^{-5} during training. As for ALuMA, 16 hypotheses are sampled from the version space via independent hit-and-run chains of length 2000. Additionally, our OptVS algorithm samples 25 hypotheses from a single hit-and-run chain, using a 100 steps burn-in and 100 steps thinning. In OptVS, we also added a small jitter of 10^{-12} to the diagonal of the kernel matrix, making it positive-definite. Additionally, we chose a rounding parameter $\gamma = 1/(n+1)\sqrt{n}$, n being the dimension of W_t.

Servers: Our experiments were run on four servers, each with 40-core Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, 128GB memory, Python 3.7 on CentOS 7.

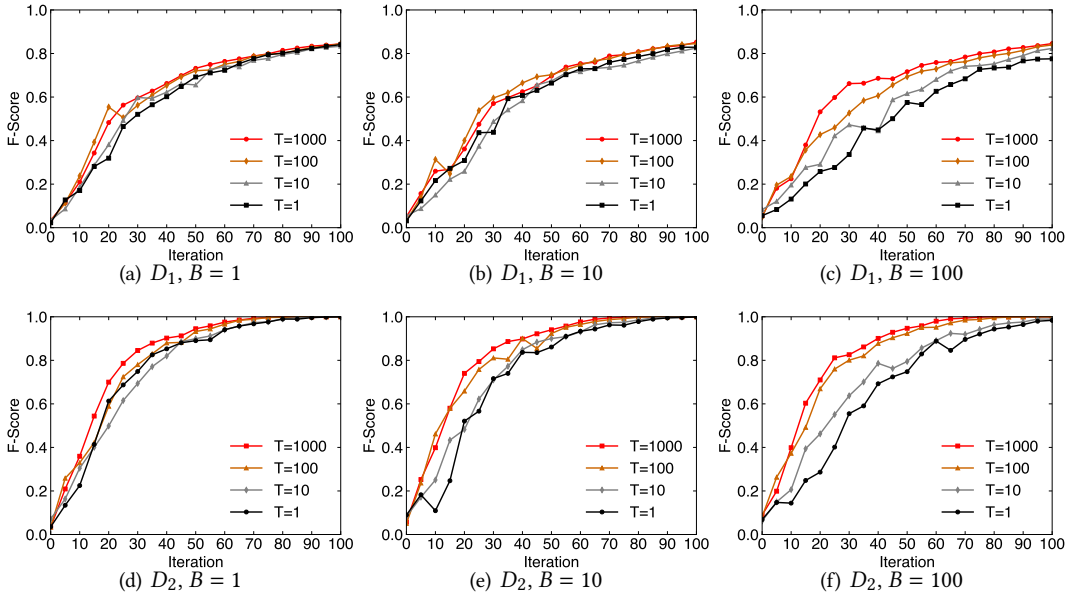
6.2 Sensitivity Study for OptVS

In this first experimental section, we perform a simple sensitivity analysis to understand how changes in OptVS' most important hyperparameters affect its performance and attempt to recommend reasonable default values. In particular, we will focus on the Hit-and-Run sampling parameters described in Section 4: the *number of burn-in iterations* B, the *number of thinning iterations* T, and the *number of samples* S.

Expt 1 (Sensitivity Analysis): For this study we generated two synthetic data sets, D_1 and D_2, based of SDSS 01 and 02 queries. Each data set consists of $N = 10,000$ data points sampled uniformly over $[-\sqrt{3}, \sqrt{3}]^4$. The positive region consists of a centered hyper-rectangle $[-L, L]^4$ in the case of D_1 and a centered ball $B(0, R)$ in the case of D_2, with L and R chosen to contain 1% of all data. Otherwise, the experimental conditions are identical as described above. We use synthetic data sets for performance reasons: running OptVS for all SDSS queries would quickly become intractable due to the large size of this data set and the high number of parameter combinations.

We begin by studying the impact of the thinning parameter T on performance, as displayed in Figure 4. First, we note that increasing T tends to improve the convergence speed, irrespective of the burn-in value B. For example, Figures 4(c) and 4(f) show the case of $B = 100$, in which we observe a difference of 5 to 20 F-Score points between the lines of smallest and largest T values. Additionally, we also observe that T has less of an impact for smaller values of B. For example, in the extreme case of $B = 1$ showcased by Figures 4(a) and 4(d), we only observe a performance difference of at most 5 F-Score points across all thinning choices.

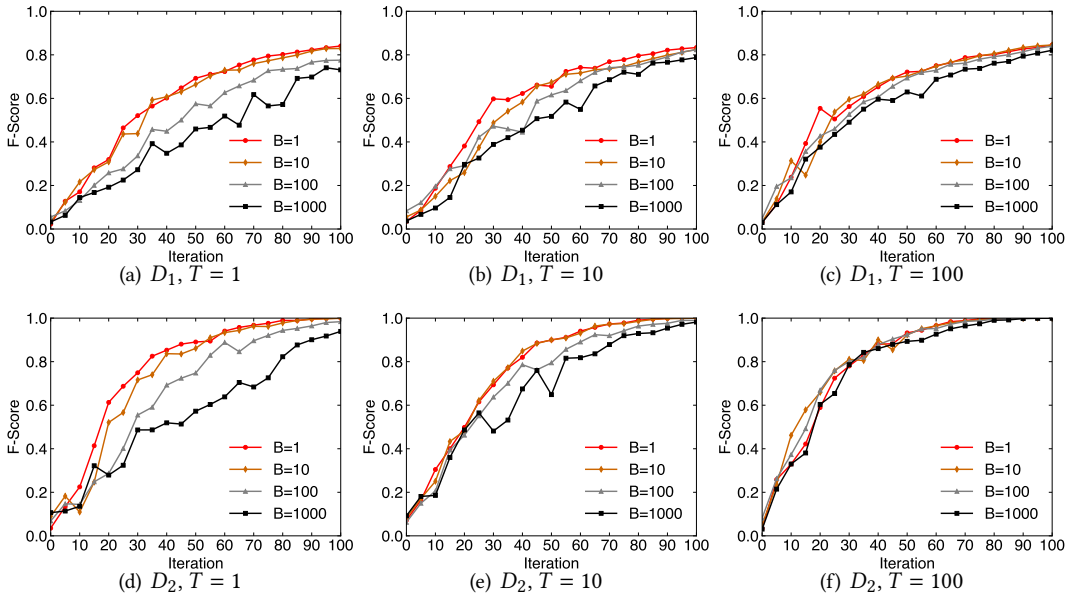
Next, we observe the effect of the burn-in parameter B as displayed in Figure 5. First, we note that *smaller* values of B tend to outperform larger values: For example, Figures 5(a) and 5(d) show an improvement of up to 40 F-Score points between the $B = 1$ and $B = 1000$ lines. In addition,



(a) $D_1, B = 1$ (b) $D_1, B = 10$ (c) $D_1, B = 100$

(d) $D_2, B = 1$ (e) $D_2, B = 10$ (f) $D_2, B = 100$

Fig. 4. Comparison of OptVS performance in terms of F-Score when varying the thinning T from 1 to 1000 while fixing the burn-in B. In all plots, the number of samples S was fixed to 16.



(a) $D_1, T = 1$ (b) $D_1, T = 10$ (c) $D_1, T = 100$

(d) $D_2, T = 1$ (e) $D_2, T = 10$ (f) $D_2, T = 100$

Fig. 5. Comparison of OptVS performance in terms of F-Score when varying the burn-in B from 1 to 1000 while fixing the thinning T. In all plots, the number of samples S was fixed to 16.

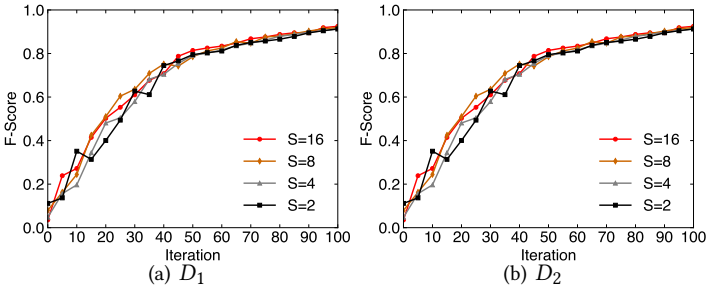


Fig. 6. Comparison of OptVS performance in terms of F-Score when varying the number of samples from 2 to 16. In all plots, the burn-in and thinning parameters were fixed to $B = 1$ and $T = 100$.

we observe that the burn-in parameter tends to have a reduced impact for larger values of T. By observing Figures 5(c) and 5(f) for example, we see that the performance between different parameters does not differ by more than 10 F-Score points across all iterations.

Finally, Figure 6 displays the results for varying the number of samples S after fixing T and B to the best values found in the previous experiments. Overall, increasing the number of samples S seems to provide a small improvement in performance, though much smaller than observed for B or T; in fact, for both data sets we do not observe a difference larger than 10 F-Score points across all iterations.

In conclusion, we can summarize our observations as follows:

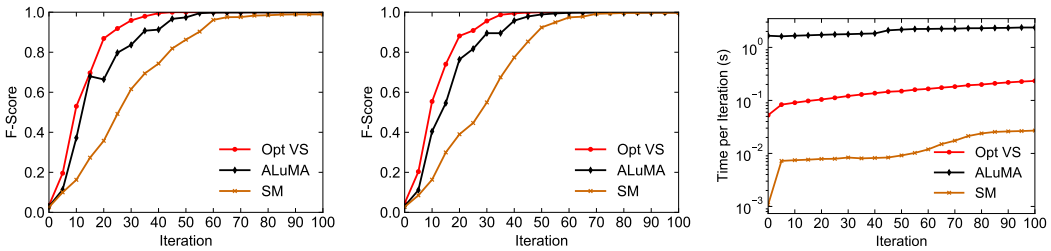
- *Thinning can substantially improve convergence speed*: Thinning T was shown to provide substantial performance improvements, although these gains tend to reduce after a sufficiently large number. We generally recommend using a value of $T = 100$ or larger.
- *Burn-in is generally* **not** *helpful*: In all cases considered, adding a burn-in B tends to have a smaller impact on performance, especially for larger values of T recommended above. For these reasons, one can use any small value for B.
- *The number of samples has minimal impact on performance*: We have not observed any significant improvements in performance by increasing S by more than 4, which seems like a good starting value to use.

The result of this sensitivity study also confirmed the result of our hyper-parameter tuning of OptVS, which was used to report its results in the following experiments: indeed, we used $T = 100$ while the other two parameters had only a limited impact on the overall performance.

6.3 Comparison to ALuMA

One major limitation of the ALuMA [15] algorithm is its costly preprocessing step: to support kernels, it requires computing the kernel matrix over the entire data set before applying a dimensionality reduction technique, a process that is very memory intensive. For this reason, we could not run this preprocessing for the SDSS data set of 1.9M data points since a $10^6 \times 10^6$ matrix would consume terabytes of memory.

Expt 2 (Comparison to ALuMA): To enable a more complete comparison with ALuMA, additionally to the Car user patterns we also generate a synthetic data set corresponding to a subsampled version of SDSS 02. More precisely, our synthetic data is composed of $N \in \{1 \times 10^4, 2 \times 10^4, 4 \times 10^4\}$ data points sampled uniformly over $[-\sqrt{3}, \sqrt{3}]^2$, with the user interest region being a centered ball containing 1% of the data. In all considered cases, after applying the ALuMA



(a) 10k data points, F-score (b) 40k data points, F-score (c) 40k data points, Time

Fig. 7. Comparison of OptVS with ALuMA [15] and Simple Margin [37] in terms of classification accuracy and time per iteration using a synthetic data set of various sizes.

preprocessing, the transformed data set has a dimensionality close to 300. We also note that while ALuMA is run over this transformed data set, all other algorithms use the original data.

We start by observing the classification accuracy results of Figures 7 and 8. In all considered cases, OptVS outperforms ALuMA at every iteration. Figures 8(a) and 8(c) show the results for two particular Car queries. In each case, our algorithm manages to reach perfect accuracy within nearly 50 labeling iterations, with ALuMA still being at 80% and all other algorithms at 70% or below. Additionally, Table 8 shows the complete set of results over all Car queries; over there, we can see that OptVS offers a much faster convergence speed than ALuMA in earlier iterations, being, on average, 20 F-score points higher across iterations 10 and 20.

Next, we observe how the algorithms compare in terms of time per iteration. Plot 7(c) displays the time measurements over the most expensive synthetic query, while Figures 8(e)–8(f) contain the two most expensive Car patterns. In all cases, ALuMA is by far the most expensive algorithm, being around 10 times slower than OptVS. We also note that Simple Margin is the fastest algorithm due to its efficient margin-based selection strategy.

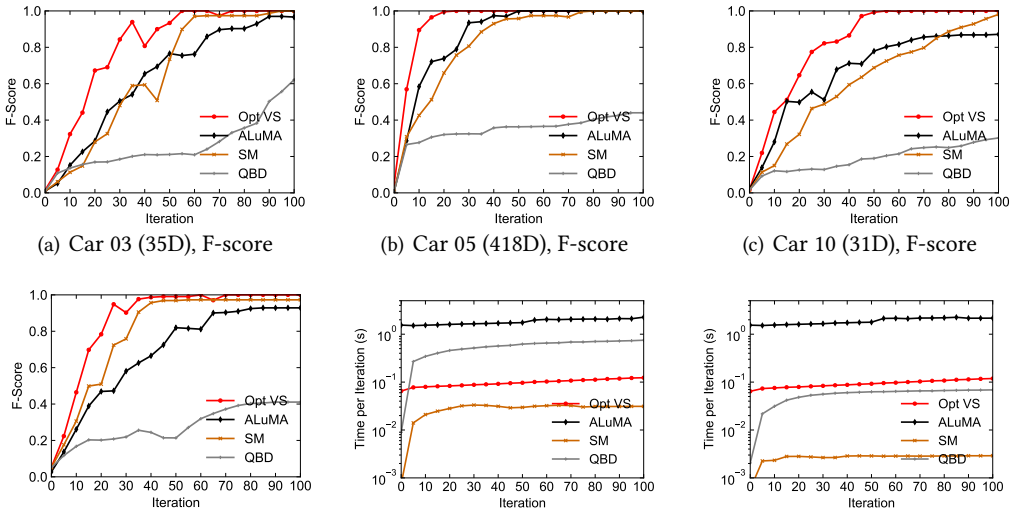
In summary, we can draw the following conclusions:

- *ALuMA preprocessing*: ALuMA's preprocessing step is very memory intensive and could not be run beyond 40k data points. In contrast, OptVS has no such restrictions.
- *Accuracy Improvement*: In all data sets considered, OptVS outperforms ALuMA at every iteration; for instance, over Car queries, we observed an improvement of 20 F-score points in average over the initial iterations. Despite employing similar exploration strategies, the difference in performance can be explained by two main factors: the use of *rounding* (Section 4.3) to improve the version space sample quality and the high-dimensionality of data after ALuMA's preprocessing (around 300).
- *Efficiency*: In all observed cases, our OptVS algorithm runs around 0.1 seconds per iteration at all times, while ALuMA takes close to 1 second. The performance benefits of our algorithm come mainly from the single-chain sampling optimization of Section 4.2.

Due to the above, we will leave ALuMA out of consideration in the remaining experimental evaluations.

6.4 Evaluating Our Techniques Using SDSS

Expt 3 (Factorization): We next study the effect of factorization by comparing OptVS with its extension to factorization, Fact VS. For factorization, each predicate in the target query corresponds



(a) Car 03 (35D), F-score (b) Car 05 (418D), F-score (c) Car 10 (31D), F-score

(d) Car 13 (24D), F-score (e) Car 05 (418D), Time (f) Car 13 (24D), Time

Fig. 8. Comparison of OptVS with other active learning techniques [15, 33, 37] in terms of classification accuracy and time per iteration using the Car data set. In addition, we display in parenthesis the dimensionality of each query after one-hot encoding.

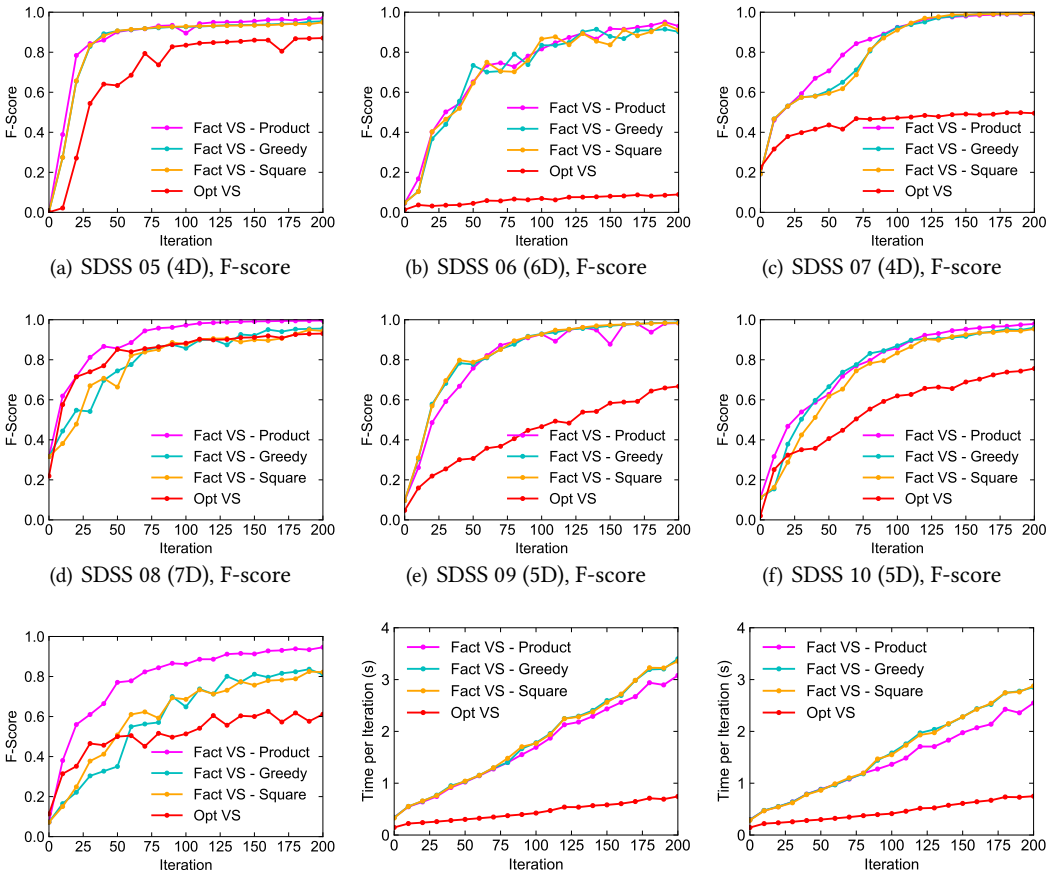
to its own factorized subspace. 01–04 are not factorized because they are 2D queries. We also note that 05–07 present no overlap of attributes across subspaces, while 08–11 present such an overlap.

For SDSS 05–11, Fact VS outperforms OptVS by a wide margin as displayed in Figures 9(a)-9(g). In all cases, factorization provides a significant boost in performance; for 06 in particular, the factorized version reaches $>$ 90% F-score after 200 iterations while the non-factorized version is still at $<$ 10%. Also, we do not observe any performance difference between the greedy and squared-loss strategies of factorization, despite the latter having a worse theoretical bound. As for the product loss, results are nearly identical to the other two losses for most queries, except 08 and 11 for which the product loss performs significantly better. This confirms our intuition of Section 5.5 that the product loss should converge faster since it attempts to directly reduce the number of final classifiers instead of the factorized ones.

Finally, the running time of Fact VS is somewhat higher than OptVS due to the handling of multiple subspaces and our current implementation that runs subspatial computations serially. Figures 9(h) and 9(i) show the time for SDSS 08 and 10, which are the two most expensive queries with 5 and 6 subspaces. The time per iteration is always below 4 seconds and, hence, satisfies the requirement of interactive speed. Note that the time per iteration can be further improved if subspatial computations are run concurrently, which is left to future work.

6.5 Comparing to Other Techniques Using SDSS

Expt 4 (Comparison to other algorithms): We next compare our algorithms to three active learning strategies: Simple Margin (SM) [37], Query-by-Disagreement (QBD) [33], and a factorization-aware algorithm, DSM [18]. Our Factorized Version Space algorithm uses the product loss, which tends to give better results. Figure 10 illustrates per-iteration measurements for a subset of queries, while Tables 5–7 show a summary of our results.



(a) SDSS 05 (4D), F-score (b) SDSS 06 (6D), F-score (c) SDSS 07 (4D), F-score

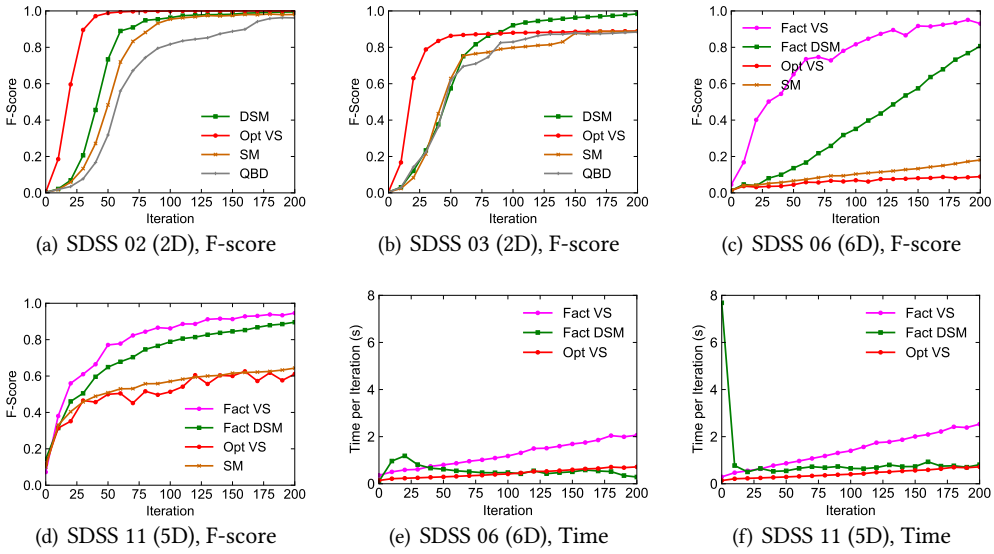
(d) SDSS 08 (7D), F-score (e) SDSS 09 (5D), F-score (f) SDSS 10 (5D), F-score

(g) SDSS 11 (5D), F-score (h) SDSS 08 (7D), Time (i) SDSS 10 (5D), Time

Fig. 9. Effects of factorization on performance using the SDSS data set. We also display in parenthesis the dimensionality of each query.

We first consider the case without factorization. As Table 5 shows, our OptVS algorithm outperforms the two VS algorithms across the majority of queries, including DSM most time over SDSS 01–04. For example, let's consider Figures 10(a) and 10(b) showing the results for SDSS 02 and 03. During the initial iterations, OptVS improves much faster than other algorithms and remains the best across all iterations of 02 and in most iterations of 03. We also observe that DSM eventually surpasses kernel-based methods such as OptVS for rectangular patterns such as SDSS 01 and 03. This difference comes from the different classification models being used: DSM's polytope model can easily capture such polygonal patterns, while the round-shaped RBF kernel will have trouble learning the corners.

Tables 6 and 7 also display the results for the precision and recall metrics. In terms of precision, we observe that OptVS outperforms other AL methods in all SDSS queries. In particular, this difference can be especially large in early iterations, with OptVS outperforming SM and QBD by at least 50 points in the first 50 iterations of SDSS 01–03. In terms of recall, however, we observe the inverse behavior with QBD and SM scoring between 10 to 20 points higher in several cases during



(a) SDSS 02 (2D), F-score (b) SDSS 03 (2D), F-score (c) SDSS 06 (6D), F-score

(d) SDSS 11 (5D), F-score (e) SDSS 06 (6D), Time (f) SDSS 11 (5D), Time

Fig. 10. Comparison of OptVS and Fact VS with other active learning techniques [18, 37] in terms of classification accuracy and time per iteration using the SDSS data set. We also display in parenthesis the dimensionality of each query.

Query #	01			02			03			04			05			06		
Fact VS	–	–	–	–	–	–	–	–	–	–	–	–	**82**	**90**	**90**	**35**	**65**	**79**
DSM	10	59	**93**	11	73	96	19	57	**92**	26	80	98	6	43	87	6	14	35
Opt VS	**81**	**90**	90	**83**	**99**	**100**	**67**	**86**	88	**97**	**100**	**100**	34	72	82	3	4	6
SM	8	42	90	9	48	95	15	63	80	86	98	**100**	3	37	74	5	7	10
QBD	4	26	81	5	32	82	18	62	83	91	96	97	0	2	24	4	4	5

Query #	07			08			09			10			11		
Fact VS	56	**71**	**92**	73	86	**97**	**54**	**76**	**93**	50	63	86	**62**	**77**	**86**
DSM	**62**	70	65	76	**87**	94	43	68	87	**55**	**76**	**89**	50	65	79
Opt VS	29	36	40	**77**	83	89	23	37	50	32	44	60	36	40	58
SM	44	48	51	70	81	86	32	37	47	34	50	59	43	51	57
QBD	41	46	54	64	78	86	27	35	47	29	39	53	34	45	59

Table 5. F-score (in %) comparison at iterations 25, 50, and 100 of OptVS and Fact VS with other active learning techniques [18, 33, 37] using the SDSS data set. Bold values represent the observed maximum across all algorithms per query at the given iteration.

earlier iterations, although this difference tends to decrease as more points are labeled. From this, we can conclude that OptVS tends to be more "conservative" in its estimates of the positive region and incrementally expands its boundaries as more points are labeled. On the other hand, SM and QBD tend to "overshoot" the positive region and gradually shrink it via new labeled examples.

Query #	01			02			03			04			05			06		
Fact VS	–	–	–	–	–	–	–	–	–	–	–	–	**86**	**93**	**90**	**69**	79	88
DSM	5	49	**94**	6	69	96	12	55	**96**	59	99	**100**	4	35	88	65	**90**	**97**
Opt VS	**79**	**90**	91	**86**	**99**	**100**	**67**	**90**	91	98	**100**	**100**	38	65	87	65	81	87
SM	4	31	90	5	40	95	9	55	80	**99**	**100**	**100**	1	25	74	51	65	82
QBD	2	16	79	2	21	78	10	51	88	97	97	96	0	1	14	4	5	15

Query #	07			08			09			10			11		
Fact VS	**81**	**84**	**96**	**81**	82	**97**	**77**	89	94	**79**	82	91	**83**	85	90
DSM	76	81	71	79	**88**	94	70	**91**	**98**	70	**85**	**93**	77	**92**	**96**
Opt VS	64	65	64	**81**	87	89	48	66	65	58	60	75	79	69	63
SM	52	47	52	68	81	86	32	48	62	32	50	62	60	69	80
QBD	44	55	59	63	81	87	21	39	54	25	40	54	50	53	69

Table 6. Precision (in %) comparison at iterations 25, 50, and 100 of OptVS and Fact VS with other active learning techniques [18, 33, 37] using the SDSS data set. Bold values represent the observed maximum across all algorithms per query at the given iteration.

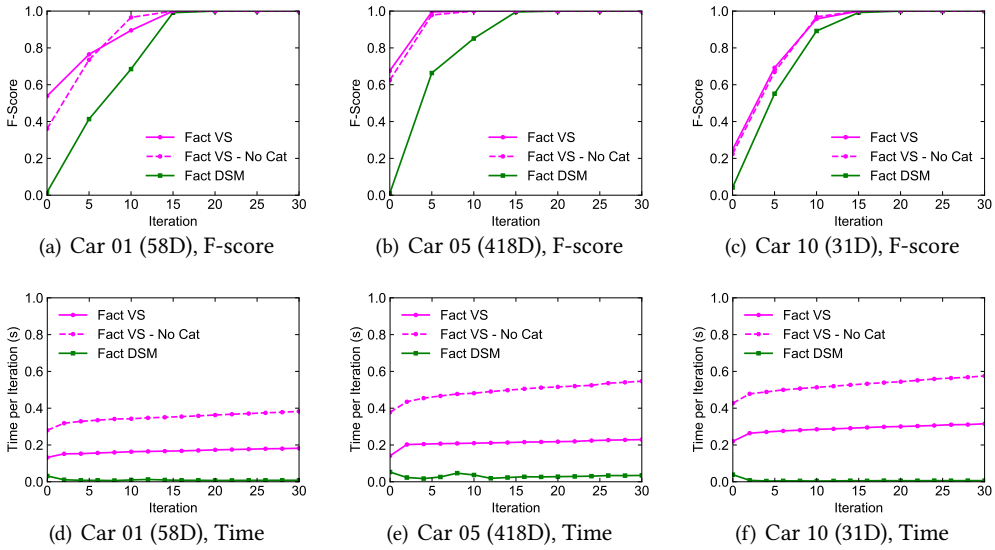
Query #	01			02			03			04			05			06		
Fact VS	–	–	–	–	–	–	–	–	–	–	–	–	79	88	**92**	**39**	**62**	**79**
DSM	97	**90**	**93**	89	90	97	**80**	78	**89**	31	69	97	82	65	85	3	8	22
Opt VS	84	89	90	82	**99**	**100**	74	83	86	**96**	**100**	**100**	50	69	80	2	2	4
SM	84	88	89	91	88	96	72	**86**	88	77	97	**100**	74	79	75	3	4	6
QBD	**100**	82	87	**100**	84	92	77	**86**	79	87	94	98	**100**	**99**	81	6	6	3

Query #	07			08			09			10			11		
Fact VS	48	64	**90**	74	**94**	**98**	**45**	**68**	**92**	37	53	82	**51**	**71**	**83**
DSM	**59**	**67**	69	76	87	94	35	56	79	**49**	**69**	**86**	39	52	68
Opt VS	36	37	42	**79**	84	88	21	24	38	23	35	54	35	48	55
SM	49	55	55	75	82	88	36	34	39	44	52	57	34	42	46
QBD	49	46	53	69	76	85	43	37	43	43	47	55	31	42	53

Table 7. Recall (in %) comparison at iterations 25, 50, and 100 of OptVS and Fact VS with other active learning techniques [18, 33, 37] using the SDSS data set. Bold values represent the observed maximum across all algorithms per query at the given iteration.

We next consider factorization. As Tables 5–7 show, our Fact VS outperforms others in the majority of cases for all metrics considered, including DSM that uses factorization and stronger assumptions. Figures 10(c) and 10(d) show the results for 06 and 11. In general, Fact VS outperforms DSM, which in turn is an upper bound of all other non-factorized algorithms. In the particular case of 10(c), after 100 iterations Fact VS is at > 80% accuracy, while DSM is still at 40%, and all of the other alternatives are at 10% or lower.

Finally, Figures 10(e) and 10(f) show the running time of Fact VS, DSM, and OptVS for SDSS 06 and 11. In both cases, the two factorized algorithms take at most a couple of seconds per iteration, thus being compatible in the interactive data exploration scenario. Moreover, we notice that DSM



(a) Car 01 (58D), F-score (b) Car 05 (418D), F-score (c) Car 10 (31D), F-score

(d) Car 01 (58D), Time (e) Car 05 (418D), Time (f) Car 10 (31D), Time

Fig. 11. Evaluating our optimization for categorical subspaces using the Car data set. We also display in parenthesis the dimensionality of the query after one-hot encoding.

has a large warm-up time, being slower than Fact VS during the first 30 iterations. Thus, Fact VS may be preferred to DSM given its better accuracy and lower time per iteration in the initial iterations.

6.6 Comparing to Other Techniques Using Car Queries

This section has two main purposes: First, we confirm that our previous conclusions over the SDSS dataset also hold over Cars queries. Second, we also demonstrate the benefits of the categorical subspace optimization in Section 5.6, which manages to improve performance while speeding up computation.

For the 18 queries over the Car data set, we do not observe a big performance difference between our factorized VS algorithm (with product loss) and DSM; as shown in Table 8, in all cases, both algorithms reach 100% accuracy in less than 20 iterations. This fast convergence is due to the data sparsity, which allows more freedom in separating the positive and negative classes.

We also note that the new results for the precision and recall metrics are consistent with the SDSS data set results: as shown in Tables 9 and 10, OptVS outperforms other non-factorized algorithms in precision for almost every case while often underperforming in recall during early iterations. Finally, we once again observe that factorized methods usually outperform non-factorized ones by a large margin in all metrics considered.

Expt 5 (Optimization for categorical attributes): We next study the effect of our optimization for categorical variables, as described in Section 5.6. Figure 11 shows the result for the three Car queries with the highest number of categorical subspaces. The "No Cat" version of our algorithm relies on the one-hot encoding of the data. As we can see, this optimization does not have a visible impact on accuracy but it can significantly reduce the time per iteration, providing a speed-up of at least 100% for all these queries. We also observed similar results over all other car queries.

Query #	01			02			03			04			05			06		
Fact VS	**76**	**90**	**100**	**96**	**100**	**100**	**67**	**86**	**100**	**65**	**90**	**100**	**100**	**100**	**100**	**84**	**100**	**100**
DSM	41	69	**100**	64	97	**100**	15	40	88	50	84	**100**	66	85	**100**	73	**100**	**100**
Opt VS	30	56	79	26	61	91	13	32	67	11	33	48	57	89	**100**	52	90	**100**
ALuMA	22	39	51	13	33	64	5	15	29	6	15	29	29	58	74	57	80	**100**
SM	15	29	60	12	18	57	6	11	28	4	9	27	31	43	66	39	73	**100**
QBD	25	29	31	8	9	13	11	14	17	5	7	8	27	28	32	28	47	48

Query #	07			08			09			10			11			12		
Fact VS	**75**	89	**100**	**100**	**100**	**100**	89	**100**	**100**	**69**	**96**	**100**	**69**	**97**	**100**	**71**	**98**	**100**
DSM	56	**90**	**100**	**100**	**100**	**100**	**92**	**100**	**100**	55	89	**100**	45	94	**100**	53	93	**100**
Opt VS	45	72	77	95	**100**	**100**	61	76	98	22	44	65	10	33	52	16	51	80
ALuMA	31	40	59	81	99	**100**	45	56	74	14	28	50	6	19	39	5	10	25
SM	18	38	56	**100**	**100**	**100**	51	71	99	11	15	32	5	7	24	6	16	58
QBD	17	21	32	52	56	67	37	38	39	9	12	13	3	4	7	4	6	9

Query #	13			14			15			16			17			18		
Fact VS	**90**	99	**100**	**94**	**100**	**100**	65	87	**100**	**99**	**100**	**100**	**35**	**56**	**100**	**98**	**100**	**100**
DSM	83	**100**	**100**	84	**100**	**100**	**70**	**93**	**100**	94	**100**	**100**	31	51	**100**	97	**100**	**100**
Opt VS	22	46	78	33	51	94	45	38	73	50	86	**100**	12	27	37	64	94	91
ALuMA	14	26	47	35	45	85	24	32	64	28	40	76	8	25	28	28	69	87
SM	18	31	51	30	41	65	24	36	66	35	53	78	5	11	22	57	82	96
QBD	12	17	20	19	24	28	35	41	46	26	29	34	8	10	12	30	38	50

Table 8. F-score (in %) comparison at iterations 5, 10, and 20 of OptVS and Fact VS with other active learning techniques [18, 33, 37] using the Car data set. Bold values represent the observed maximum across all algorithms per query at the given iteration.

6.7 Summary of Experimental Results

From the previous experiments, we can draw the following conclusions regarding our proposed algorithms:

1. *Accuracy Improvement (OptVS)*: OptVS was observed to outperform other VS algorithms in all Car queries and 6 out of 11 SDSS queries while still achieving comparable performance in the remaining cases. In particular, we note that OptVS can reach high accuracy with a small number of labeled points: in the case of SDSS 01–03, OptVS reached an accuracy larger than 67% after only 25 labeling iterations, while all other VS algorithms were still below 19% F-score. Besides, OptVS was also shown to outperform DSM in non-factorized scenarios.

2. *Advantages of factorization*: Fact VS was shown to outperform OptVS in almost all high-dimensional cases, irrespective of the strategy being used. Furthermore, we did not observe any performance difference between the greedy and the squared-loss strategies, but we did observe a few cases in which the product loss significantly outperforms the other two.

3. *Accuracy Improvement (Fact VS)*: For higher dimensional problems, our Fact VS algorithm significantly outperforms other VS algorithms as well as DSM, a factorization-aware algorithm, while maintaining interactive speed. For example, for a complex user interest pattern tested, our algorithm achieves an F-score of over 80% after 100 iterations of labeling, while DSM is still at 40% and all other VS algorithms are at 10% or lower. The difference between Fact VS and DSM is reduced for the simple Car data set due to the data sparsity and, consequently, fast convergence.

4. *Labeled set size*: Both OptVS and Fact VS algorithms can achieve high classification accuracy while only requiring a relatively small number of examples to be labeled. Taking the Cars queries

Query #	01			02			03			04			05			06		
Fact VS	**75**	**94**	**100**	**100**	**100**	**100**	**55**	**84**	**100**	**56**	**88**	**100**	**99**	**100**	**100**	**81**	**100**	**100**
DSM	28	60	**100**	60	97	**100**	8	30	92	43	83	**100**	51	84	**100**	68	**100**	**100**
Opt VS	23	52	93	20	62	90	15	45	93	6	34	56	44	85	99	62	94	**100**
ALuMA	14	36	58	7	25	80	3	12	42	3	9	22	17	43	65	62	83	**100**
SM	9	21	46	6	11	51	3	6	30	2	5	17	19	27	53	29	66	**100**
QBD	15	18	21	4	5	7	6	8	10	3	4	4	16	16	19	18	33	35

Query #	07			08			09			10			11			12		
Fact VS	**81**	**95**	**100**	**100**	**100**	**100**	83	**100**	**100**	**65**	**93**	**100**	**69**	**99**	**100**	**67**	**100**	**100**
DSM	45	87	**100**	**100**	**100**	**100**	**88**	**100**	**100**	44	90	**100**	39	92	**100**	42	90	**100**
Opt VS	40	83	86	94	**100**	**100**	44	65	97	14	68	82	6	42	66	14	69	92
ALuMA	31	63	78	73	99	**100**	29	40	65	8	24	50	3	13	47	2	6	20
SM	12	32	47	99	**100**	**100**	34	55	97	6	9	23	2	4	18	3	11	50
QBD	10	14	23	35	40	55	23	24	24	5	7	7	2	2	4	2	3	5

Query #	13			14			15			16			17			18		
Fact VS	**89**	98	**100**	**91**	**100**	**100**	**66**	86	**100**	**98**	**100**	**100**	**33**	**62**	99	**99**	**100**	**100**
DSM	75	**100**	**100**	81	99	**100**	64	**96**	**100**	90	**100**	**100**	22	51	**100**	98	**100**	**100**
Opt VS	20	74	92	22	57	96	48	81	96	35	82	**100**	7	29	52	69	88	89
ALuMA	8	20	70	23	39	83	20	54	91	16	25	69	4	20	35	19	71	86
SM	10	19	38	19	36	56	15	28	66	21	37	66	3	7	17	42	80	94
QBD	6	10	12	11	14	16	24	32	38	15	17	21	4	6	7	22	30	39

Table 9. Precision (in %) comparison at iterations 5, 10, and 20 of OptVS and Fact VS with other active learning techniques [18, 33, 37] using the Car data set. Bold values represent the observed maximum across all algorithms per query at the given iteration.

as an example, with a budget of only 20 labeled data points, Fact VS obtains perfect classification accuracy while OptVS reaches 65% F-Score in 15 out of 18 cases. Similarly, given a budget of 100 labeled examples over SDSS patterns, Fact VS reaches at least 80% F-Score in all cases while OptVS does the same for 6 out of 11 cases.

5. *Precision and Recall*: In the non-factorized case, OptVS was shown to outperform other techniques in precision in almost all cases considered while often slightly underperforming in recall during early iterations. In the particular cases of SDSS 01–03 for example, OptVS scores up to 50 points higher than SM and QBD in precision during early iterations while maintaining a gap of at most 10 recall points. We anticipate that the significant gains in precision outweigh a modest loss in recall in our target application of interactive data exploration. Furthermore, when factorization is applied, we observe that factorized methods usually score much higher than non-factorized algorithms in both precision and recall.

6. *Efficiency*: OptVS runs under interactive performance at all times, taking less than 1 second per iteration and being around 10 times faster than ALuMA. We also note that SM usually has the fastest per-iteration time due to the efficient implementation of existing SVM solvers, but at the cost of overall limited accuracy. In the factorized cases, we have observed that Fact VS tends to be faster than DSM in initial labeling iterations while still running under 4 seconds per iteration at all times. Compared to non-factorized algorithms, we observe that factorized methods are generally slower due to the need of processing multiple subspaces per iteration, which could be further improved by leveraging parallel computation as we discussed in Section 5.7.

Query #	01			02			03			04			05			06		
Fact VS	83	88	**100**	92	**100**	**100**	**91**	**95**	**100**	82	93	**100**	**100**	**100**	**100**	92	**100**	**100**
DSM	81	**89**	**100**	78	97	**100**	79	73	89	89	86	**100**	**100**	91	**100**	88	**100**	**100**
Opt VS	64	75	72	65	69	93	52	40	59	79	52	55	**100**	96	**100**	64	88	**100**
ALuMA	69	62	63	80	54	58	54	42	28	86	71	56	**100**	**100**	91	72	84	**100**
SM	**90**	70	94	89	65	79	82	65	56	**100**	91	82	**100**	**100**	98	76	88	**100**
QBD	87	80	77	**100**	**100**	**100**	77	73	68	**100**	**100**	98	**100**	**100**	**100**	**100**	98	84

Query #	07			08			09			10			11			12		
Fact VS	75	86	**100**	**100**	**100**	**100**	98	**100**	**100**	82	**100**	**100**	85	95	**100**	95	96	**100**
DSM	**90**	**95**	**100**	**100**	**100**	**100**	98	**100**	**100**	**84**	90	**100**	68	**97**	**100**	96	**97**	**100**
Opt VS	73	69	80	**100**	**100**	**100**	**100**	98	**100**	68	48	64	68	44	53	69	47	75
ALuMA	46	31	58	**100**	**100**	**100**	**100**	**100**	91	59	58	64	57	48	43	86	58	51
SM	85	85	89	**100**	**100**	**100**	**100**	**100**	**100**	78	68	66	**91**	60	64	91	73	75
QBD	57	57	57	**100**	**100**	**100**	**100**	**100**	**100**	**84**	81	60	68	60	58	**98**	87	78

Query #	13			14			15			16			17			18		
Fact VS	94	**100**	**100**	99	**100**	**100**	76	**95**	**100**	99	**100**	**100**	55	**71**	**100**	97	**100**	**100**
DSM	98	**100**	**100**	92	**100**	**100**	82	90	**100**	**100**	**100**	**100**	**62**	64	**100**	96	**100**	**100**
Opt VS	48	46	78	80	72	92	51	29	64	**100**	96	**100**	47	36	34	68	**100**	**100**
ALuMA	73	58	52	80	73	91	42	35	53	**100**	**100**	96	49	45	36	79	77	94
SM	**100**	88	85	88	77	88	**84**	60	74	**100**	**100**	**100**	60	50	42	**100**	90	99
QBD	**100**	98	90	**100**	98	98	78	66	63	**100**	**100**	**100**	59	55	50	92	92	94

Table 10. Recall (in %) comparison at iterations 5, 10, and 20 of OptVS and Fact VS with other active learning techniques [18, 33, 37] using the Car data set. Bold values represent the observed maximum across all algorithms per query at the given iteration.

6. *Optimization for categorical subspaces*: Our optimization for categorical subspaces was shown to significantly improve the running time of our algorithms without compromising the accuracy, leading to improvements of up to 100% in running time.

7 CONCLUSIONS

To overcome the slow start problem of active learning (AL) for model development over large datasets, we presented three major contributions: (1) a novel theoretical framework for version-space-based AL algorithms over kernel classifiers, with strong theoretical guarantees on performance and optimizations for large data sets; (2) an efficient algorithm implementing our proposed strategy, introducing several optimizations for sampling the version space; and (3) an extended version space algorithm leveraging a factorization structure provided by the user to perform active learning in a set of subspaces. Using real-world data sets, our evaluation results show that our algorithms significantly outperform state-of-the-art version space techniques, including Simple Margin [37], Query-by-Disagreement [33], and ALuMA [15], as well as a factorization-aware algorithm, DSM [18], while maintaining interactive speed.

For future work, one direction is to extend our version-space-based techniques to handle noisy labels, which are likely to occur in real applications. Another line of work is to extend the factorization technique to other classes of Active Learning algorithms, such as Uncertainty Sampling or Expected Error Reduction methods. Furthermore, we will explore how the interactive data exploration framework could be integrated with Data Programming [2, 31] for efficient model

development, combining our fast convergence in early iterations of human labeling with the auto-labeling functionality of Data Programming to scale to large datasets.

REFERENCES

[1] Francis R. Bach and Michael I. Jordan. 2005. Predictive Low-Rank Decomposition for Kernel Methods. In *International Conference on Machine Learning (ICML)*. Association for Computing Machinery, New York, NY, USA, 33–40. https://doi.org/10.1145/1102351.1102356

[2] Stephen H. Bach, Bryan He, Alexander Ratner, and Christopher Ré. 2017. Learning the Structure of Generative Models without Labeled Data. In *International Conference on Machine Learning*, Vol. 70. 273–282.

[3] Claude J. P. Bélisle, H. Edwin Romeijn, and Robert L. Smith. 1993. Hit-and-Run Algorithms for Generating Multivariate Distributions. *Mathematics of Operations Research* 18, 2 (1993), 255–266.

[4] Robert G. Bland, Donald Goldfarb, and Michael J. Todd. 1981. Feature Article—The Ellipsoid Method: A Survey. *Operations Research* 29, 6 (1981), 1039–1091. https://doi.org/10.1287/opre.29.6.1039

[5] Lawrence Bull, Keith Worden, Graeme Manson, and Nikolaos Dervilis. 2018. Active learning for semi-supervised structural health monitoring. *Journal of Sound and Vibration* 437 (2018), 373 – 388. https://doi.org/10.1016/j.jsv.2018.08.040

[6] Yuxin Chen and Andreas Krause. 2013. Near-optimal Batch Mode Active Learning and Adaptive Submodular Optimization. In *International Conference on Machine Learning*, Vol. 28. 160–168.

[7] Sanjoy Dasgupta. 2005. Analysis of a greedy active learning strategy. In *Advances in Neural Information Processing Systems 17*. 337–344.

[8] Daniele De Martino, Matteo Mori, and Valerio Parisi. 2015. Uniform sampling of steady states in metabolic networks: Heterogeneous scales and rounding. *PLoS ONE* 10, 4 (2015), e0122670. https://doi.org/10.1371/journal.pone.0122670

[9] Luciano Di Palma, Yanlei Diao, and Anna Liu. 2019. A Factorized Version Space Algorithm for "Human-In-the-Loop" Data Exploration. In *2019 IEEE International Conference on Data Mining (ICDM)*. 1018–1023. https://doi.org/10.1109/ICDM.2019.00117

[10] Kyriaki Dimitriadou, Olga Papaemanouil, and Yanlei Diao. 2016. AIDE: An Active Learning-Based Approach for Interactive Data Exploration. *TKDE* 28, 11 (2016), 2842–2856.

[11] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: an automatic query steering framework for interactive data exploration. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 517–528.

[12] Ran Gilad-Bachrach, Amir Navot, and Naftali Tishby. 2006. Query-by-Committee Made Real. In *Advances in Neural Information Processing Systems 18*. 443–450.

[13] Donald Goldfarb and Michael J. Todd. 1982. Modifications and implementation of the ellipsoid algorithm for linear programming. *Mathematical Programming* 23, 1 (1982), 1–19. https://doi.org/10.1007/BF01583776

[14] Daniel Golovin and Andreas Krause. 2011. Adaptive Submodularity: Theory and Applications in Active Learning and Stochastic Optimization. *Journal of Artificial Intelligence Research* 42 (2011), 427–486. https://doi.org/10.1613/jair.3278

[15] Alon Gonen, Sivan Sabato, and Shai Shalev-Shwartz. 2013. Efficient active learning of halfspaces: an aggressive approach. *Journal of Machine Learning Research* 14, 1 (2013), 2583–2615.

[16] Simone Hantke, Zixing Zhang, and Björn W. Schuller. 2017. Towards Intelligent Crowdsourcing for Audio Data Annotation: Integrating Active Learning in the Real World. In *INTERSPEECH*. 3951–3955.

[17] Robbert L. Harms and Alard Roebroeck. 2018. Robust and Fast Markov Chain Monte Carlo Sampling of Diffusion MRI Microstructure Models. *Frontiers in Neuroinformatics* 12 (2018), 97. https://doi.org/10.3389/fninf.2018.00097

[18] Enhui Huang, Liping Peng, Luciano Di Palma, Ahmed Abdelkafi, Anna Liu, and Yanlei Diao. 2018. Optimization for active learning-based interactive database exploration. *Proceedings of the VLDB Endowment* 12, 1 (2018), 71–84.

[19] Brendan Juba and Hai Le. 2019. Precision-Recall versus Accuracy and the Role of Large Data Sets. In *AAAI Conference on Artificial Intelligence*, Vol. 33. 4039–4048.

[20] Haya Kaspi, Sean P. Meyn, and Richard L. Tweedie. 1997. Markov Chains and Stochastic Stability. *J. Amer. Statist. Assoc.* 92, 438 (1997), 792. https://doi.org/10.2307/2965732

[21] Akshay Krishnamurthy, Alekh Agarwal, Tzu-Kuo Huang, Hal Daumé, and John Langford. 2017. Active Learning for Cost-Sensitive Classification. In *International Conference on Machine Learning*, Vol. 70. 1915–1924.

[22] William A. Link and Mitchell J. Eaton. 2012. On thinning of chains in MCMC. *Methods in Ecology and Evolution* 3, 1 (2012), 112–115. https://doi.org/10.1111/j.2041-210X.2011.00131.x

[23] Wenzhao Liu, Yanlei Diao, and Anna Liu. 2017. An Analysis of Query-Agnostic Sampling for Interactive Data Exploration. *Communications in Statistics – Theory and Methods* 47, 16 (2017), 3820–3837.

[24] Lászlo Lovász. 1986. *An Algorithmic Theory of Numbers, Graphs and Convexity*. CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 50. Society for Industrial and Applied Mathematics.

[25] László Lovász. 1999. Hit-and-run mixes fast. *Mathematical Programming* 86, 3 (1999), 443–461. https://doi.org/10.1007/s101070050099
[26] George Marsaglia. 2007. Choosing a Point from the Surface of a Sphere. *The Annals of Mathematical Statistics* 43, 2 (2007), 645–646. https://doi.org/10.1214/aoms/1177692644
[27] Tom M. Mitchell. 1982. Generalization as search. *Artificial Intelligence* 18, 2 (1982), 203–226. https://doi.org/10.1016/0004-3702(82)90040-6
[28] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2012. *Foundations of Machine Learning*. The MIT Press.
[29] Robert M. Monarch. 2021. *Human-in-the-Loop Machine Learning: Active learning and annotation for human-centered AI*. Manning Publications.
[30] Daniela Pohl, Abdelhamid Bouchachia, and Hermann Hellwagner. 2018. Batch-based active learning: Application to social media data for crisis management. *Expert Systems with Applications* 93 (2018), 232 – 244. https://doi.org/10.1016/j.eswa.2017.10.026
[31] Alexander J. Ratner, Christopher M. De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data Programming: Creating Large Training Sets, Quickly. In *Advances in Neural Information Processing Systems 29*. 3567–3575.
[32] N Sauer. 1972. On the density of families of sets. *Journal of Combinatorial Theory, Series A* 13, 1 (1972), 145–147. https://doi.org/10.1016/0097-3165(72)90019-2
[33] Burr Settles. 2012. *Active Learning*. Morgan & Claypool Publishers.
[34] Hyunjune S. Seung, Manfred Opper, and Haim Sompolinsky. 1992. Query by Committee. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. 287–294. https://doi.org/10.1145/130385.130417
[35] Alex J. Smola and Bernhard Schökopf. 2000. Sparse Greedy Matrix Approximation for Machine Learning. In *International Conference on Machine Learning (ICML)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 911–918.
[36] Jinhua Song, Hao Wang, Yang Gao, and Bo An. 2018. Active learning with confidence-based answers for crowdsourcing labeling tasks. *Knowledge-Based Systems* 159 (2018), 244 – 258. https://doi.org/10.1016/j.knosys.2018.07.010
[37] Simon Tong and Daphne Koller. 2001. Support Vector Machine Active Learning with Applications to Text Classification. *Journal of Machine Learning Research* 2 (2001), 45–66.
[38] Kirill Trapeznikov, Venkatesh Saligrama, and David Castañón. 2011. Active Boosted Learning (ActBoost). In *International Conference on Artificial Intelligence and Statistics*, Vol. 14. 743–751.
[39] Paroma Varma and Christopher Ré. 2018. Snuba: Automating Weak Supervision to Label Training Data. *Proceedings of the VLDB Endowment* 12, 3 (2018), 223–236.
[40] Lin Yang, Yizhe Zhang, Jianxu Chen, Siyuan Zhang, and Danny Chen. 2017. Suggestive Annotation: A Deep Active Learning Framework for Biomedical Image Segmentation. In *International Conference on Medical Image Computing and Computer-assisted Intervention*. 399–407.

A PROOFS

Below we outline the proofs of our main theoretical results.

A.1 Proof of Theorem 3.3

Since $S = span(\phi(x_1), \ldots, \phi(x_N))$ is a finite-dimension subspace of the Hilbert Space $\mathcal{F}$, we can write $\mathcal{F} = S \oplus S^{\perp}$ where $S^{\perp}$ is the orthogonal complement of S. In particular, this guarantees that any $f \in \mathcal{F}$ can be decomposed as

$$f = f|_S + f|_{S^{\perp}}$$

Thus, for any $x_i \in \mathcal{X}$ we have

$$\langle \phi(x_i), f \rangle_{\mathcal{F}} = \langle \phi(x_i), f|_S \rangle_{\mathcal{F}} + \langle \phi(x_i), f|_{S^{\perp}} \rangle_{\mathcal{F}} = \langle \phi(x_i), f|_S \rangle_{\mathcal{F}}$$

In conclusion, we also note that

$$h_{b,f}(x_i) = \text{sign } b + \langle \phi(x_i), f \rangle_{\mathcal{F}} = \text{sign } b + \langle \phi(x_i), f|_S \rangle_{\mathcal{F}} = h_{b,f|_S}(x_i)$$

which finishes our proof.

A.2 Proof of Theorem 3.6

Through simple calculations, we can see that

$$\begin{aligned} p_{x_j,y} &= \mathbb{P}_{(b,\alpha)\sim\pi}\left(y(b + K_j^T\alpha) \geq 0 \,\middle|\, (b,\alpha) \in \mathcal{V}_\alpha\right) \\ &= \mathbb{P}_{(b,\alpha)\sim\pi}\left(yh_{b,\alpha}(x_j) \geq 0 \,\middle|\, (b,\alpha) \in \mathcal{V}_\alpha\right) \\ &= \mathbb{P}_{(b,\alpha)\sim\pi}\left(h_{b,\alpha}(x_j) = y \,\middle|\, (b,\alpha) \in \mathcal{V}_\alpha\right) \end{aligned}$$

where in the last passage we used the fact that $y \in \{\pm 1\}$. Now, let us define $\tilde{\mathcal{V}}$ as the true version space and $\tilde{\pi}(h) = \mathbb{P}_{(b,\alpha)\sim\pi}(h_{b,\alpha} \sim h)$ as a prior probability over $\tilde{\mathcal{V}}$. Then, it is easy to see that

$$p_{x_j,y} = \mathbb{P}_{h_{b,\alpha}\sim\tilde{\pi}}\left(h_{b,\alpha}(x_j) = y \,\middle|\, h_{b,\alpha} \in \tilde{\mathcal{V}}\right)$$

With this, our result easily follows from the near-optimality result of Equation (3) for the general GBS strategy.

A.3 Proof of Lemma 3.7

The proof of this result relies on the following lemma, which is an extension of Theorem 1 from Gilad-Bachrach et al. [12]:

Lemma A.1. *Let $W = \{w \in \mathbb{R}^N : \|w\| \leq 1$ and $a_i^T w \geq 0,\ 1 \leq i \leq t\}$ and some $a^* \in \mathbb{R}^N$. Then, for any linear subspace S containing $span(a_1, \ldots, a_t, a^*)$, we have*

$$\mathbb{P}_{w\sim Unif(W)}(\pm w^T a^* \geq 0) = \mathbb{P}_{w\sim Unif(W\cap S)}(\pm w^T a^* \geq 0)$$

Proof. Let R be any rotation mapping S into $\mathbb{R}^k \times \{0\}^{N-k}$, where $k = dim(S)$. Since rotations preserve norms and scalar products, we have that

- W is mapped into $\tilde{W} = RW = \{\tilde{w} \in \mathbb{R}^N : \|\tilde{w}\| \leq 1$ and $\tilde{a}_i^T \tilde{w} \geq 0,\ 1 \leq i \leq t\}$, where $\tilde{w} = Rw$ and $\tilde{a}_i = Ra_i$
- $w^T a^* = \tilde{w}^T \tilde{a}^*$, with $\tilde{a}^* = Ra^*$
- The random variable $w \sim Unif(W)$ is mapped into $\tilde{w} = Rw \sim Unif(\tilde{W})$

From these observations, it is clear that if the lemma holds true for the subspace $\mathbb{R}^k \times \{0\}^{N-k}$, then it must also hold true for S. In what follows, we restrict ourselves to this particular case.

Now, let $U \in \mathbb{R}^k$ be uniformly distributed over $W \cap S$. Its p.d.f. is given by

$$p_U(u) \propto \mathbb{1}(u \in W \cap S) = \mathbb{1}(\|u\| \leq 1)\mathbb{1}(\tilde{a}_i^T u \geq 0,\ \forall i)$$

where $\tilde{a}_i = (a_i^1, \ldots, a_i^k)$. Now, let's consider the random vector $V \in \mathbb{R}^{N-k}$ which is sampled conditionally on $U = u$ according to the distribution

$$p_{V|U=u}(v) \propto \mathbb{1}\left(\|u\|^2 + \|v\|^2 \leq 1\right)$$

Finally, let's define the random vector $\omega = (U_1, \ldots, U_k, V_1, \ldots, V_{N-k}) \in \mathbb{R}^N$. This vector is distributed according to

$$p_\omega(w) = p_U(w_1, \ldots, w_k)P_{V|U=(w_1,\ldots,w_k)}(w_{k+1}, \ldots, w_N) \propto \mathbb{1}\left(\|w\|^2 \leq 1\right)\mathbb{1}(a_i^T w \geq 0,\ \forall i)$$

from where we can conclude that $\omega \sim Unif(W)$. With this, using the fact that $a^* \in S = \mathbb{R}^k \times \{0\}^{N-k}$ we can finally conclude that

$$\mathbb{P}_{w\sim Unif(W)}(\pm w^T a^* \geq 0) = \mathbb{P}_{U\sim Unif(W\cap S)}(\pm U^T \tilde{a}^* \geq 0)$$

which concludes our demonstration. □

With the above lemma, we can finally prove our main result. Let's define $a_i = y_i(1, L_i)$ for $1 \leq i \leq t$, $a^* = (1, L_j)$, and $w = (b, \beta)$. Then, we have that:

- $\mathcal{V}_\beta = \{w : \|w\| \leq 1 \text{ and } a_i^T w \geq 0,\ 1 \leq i \leq t\}$, which is identical to W
- $S_j = \mathbb{R} \times span(L_1, \ldots, L_t, L_j)$ clearly contains $span(a_1, \ldots, a_t, a^*)$

With this, we can apply our lemma to $\mathcal{V}_\beta$ and S_j, which implies that

$$p_{x_j,\pm} = \mathbb{P}_{w \sim Unif(\mathcal{V}_\beta)}(\pm a^* w \geq 0) = \mathbb{P}_{w' \sim Unif(\mathcal{V}_\beta \cap S_j)}(\pm a^* w' \geq 0) = \mathbb{P}(\pm(b' + L_j^T \beta') \geq 0)$$

where $w' = (b', \beta')$ is drawn uniformly over $\mathcal{V}_\beta \cap S_j$.

A.4 Proof of Lemma 3.8

Let's define $\tilde{L}_i = (0, L_i^1, \ldots, L_i^N)$. We first observe that

$$S_j = \mathbb{R} \times span(L_1, \ldots, L_t, L_j) = span(\vec{e_1}, \tilde{L}_1, \ldots, \tilde{L}_t, \tilde{L}_j)$$

Now, by using the fact $L_i^k = 0$ for $k > i$ and $L_i^i \neq 0$ for all i, we can see that

$$span(\tilde{L}_1, \ldots, \tilde{L}_t) = \{0\} \times \mathbb{R}^t \times \{0\}^{N-t-1} = span(\vec{e_2}, \ldots, \vec{e_{t+1}})$$

which implies $S_j = span(\vec{e_1}, \ldots, \vec{e_{t+1}}, \tilde{L}_j)$. Finally, the last orthonormal basis vector can be constructed by projecting $\tilde{L}_j$ onto $span(\vec{e_1}, \ldots, \vec{e_{t+1}})^\perp$:

$$\vec{T_j} = proj_{span(\vec{e_1}, \ldots, \vec{e_{t+1}})^\perp} \tilde{L}_j = (\underbrace{0, \ldots, 0}_{t+1}, L_j^{t+1}, \ldots, L_j^N)$$

and then normalizing, which concludes our demonstration.

A.5 Proof of Theorem 3.10

By Lemma 3.7, the cut probabilities satisfy $p_{x_j,+} = \mathbb{P}(b' + \beta'^T L_j \geq 0)$, where (b', β') is uniformly distributed over $\mathcal{V}_t \cap S_j$. However, (b', β') can also be written in terms of the orthonormal basis coefficients of Equation (6), resulting in

$$\begin{aligned}
&(b', \beta') \in \mathcal{V}_t \cap S_j \\
&\iff (b', \beta') = \sum_{i=1}^{t+1} w_i \vec{e_i} + w_{t+2} \frac{\vec{T_j}}{\|\vec{T_j}\|} \text{ and } b'^2 + \|\beta'\|^2 \leq 1 \text{ and } y_i(b' + L_i^T \beta') \geq 0,\ \forall i \\
&\iff \|w\|^2 \leq 1 \text{ and } y_i \left(w_1 + \sum_{k=2}^{t+1} L_i^k w_k \right) \geq 0,\ \forall i \\
&\iff \|w\|^2 \leq 1 \text{ and } y_i \ell_i^T w \geq 0,\ \forall i \\
&\iff w \in W_t
\end{aligned}$$

In particular, this implies that sampling $(b', \beta') \sim Unif(S_j \cap \mathcal{V}_t)$ is equivalent to sampling $w \sim Unif(W_t)$. Finally, we perform the same substitution on the cut probabilities and obtain

$$\begin{aligned} p_{x_j,+} &= \mathbb{P}(b' + \beta'^T L_j \geq 0) \\ &= \mathbb{P}\left(w_1 + \sum_{i=2}^{t+1} w_i \vec{e}_i^{\,T} L_j + w_{t+2} \frac{\vec{T}_j^{\,T} L_j}{\|\vec{T}_j\|} \geq 0\right) \\ &= \mathbb{P}\left(w_1 + \sum_{i=2}^{t+1} L_j^i w_i + \|\vec{T}_j\| w_{t+2} \geq 0\right) \\ &= \mathbb{P}(\ell_j^T w \geq 0) \end{aligned}$$

With this, our proof is concluded by simply noticing that

$$\|\vec{T}_j\| = \sqrt{\|L_j\|^2 - \sum_{k=1}^{t} (L_j^k)^2} = \sqrt{K_{jj} - \sum_{k=1}^{t} (L_j^k)^2}$$

A.6 Proof of Lemma 3.12

Our objective is to apply Theorem 2.2 from Mohri et al. [28], which is given as follows: Let $p(x)$ be any prior probability over a data space $\mathcal{D}$, let S be an i.i.d sample from $p(x)$ of size m, and let $\mathcal{H}'$ be a finite set of hypothesis mapping $\mathcal{D}$ into the label set $\mathcal{Y}$. Then, for any $\delta \in (0, 1)$, with probability $1 - \delta$ it holds that

$$\forall h \in \mathcal{H}',\ \mathbb{P}_X(h(X) \neq h^*(X)) \leq \epsilon_{acc}(h, S) + \sqrt{\frac{\log |\mathcal{H}'| + \log \frac{2}{\delta}}{2m}}$$

With this, our result easily follows by considering two points:

(1) First, we choose $\mathcal{D} = \mathcal{X}$ and we select the prior $p(x_i) = \frac{1}{N}$. In particular, S must be a sample with replacement from $\mathcal{X}$, and $\mathbb{P}_X(h(X) \neq Y) = \epsilon_{acc}(h, \mathcal{X})$
(2) We choose $\mathcal{H}' = \mathcal{H}/\sim$. This is possible because every $[h] \in \mathcal{H}/\sim$ can be considered as a classifier $\mathcal{X} \to \mathcal{Y}$ defined by $[h](x_i) = h(x_i), \forall i$; in addition, $\hat{\mathcal{H}}$ is finite.

Our result is finally proved by noticing that $\epsilon_{acc}(h, \cdot)$ is constant over each equivalence class $[h]$.

A.7 Proof of Theorem 3.14

For notation convenience, we will ignore the second parameter in the error functions. The first inequality is easy to prove:

$$\begin{aligned} \epsilon_{fscore}(h) &= 1 - fscore(h) \\ &= 1 - harmonic_mean(precision(h), recall(h)) \\ &\leq 1 - \min(precision(h), recall(h)) \\ &= \max(1 - precision(h), 1 - recall(h)) \\ &= \max(\epsilon_{prec}(h), \epsilon_{rec}(h)) \end{aligned}$$

The second inequality can be proved in two parts. First, Theorem 1 from Juba and Le [19] tells us that $\max(\epsilon_{prec}(h), \epsilon_{rec}(h)) \leq \frac{1}{\mu}\epsilon_{acc}(h)$ whenever $\epsilon_{prec}(h) \leq 0.5$. Second, we have that:

$$\begin{aligned}
\epsilon_{prec} \geq 0.5 &\iff \frac{C^{+,-}}{C^{+,+} + C^{+,-}} \geq 0.5 \\
&\iff C^{+,-} \geq C^{+,+} \\
&\iff C^{+,-} + C^{-,+} \geq C^{+,+} + C^{-,+} \\
&\iff \epsilon_{acc} \geq \mu
\end{aligned}$$

In other words, if $\epsilon_{prec}(h) \geq 0.5$, then $\max(\epsilon_{prec}(h), \epsilon_{rec}(h)) \leq 1 \leq \epsilon_{acc}(h)/\mu$, which concludes our proof.

A.8 Proof of Theorem 3.16

Let X be a random variable sampled uniformly at random from S. Our result is then a simple application of the Markov inequality (Equation C.13 from Mohri et al. 28):

$$\begin{aligned}
\epsilon_{acc}(MV, S) &= \mathbb{P}_X\left(MV(X) \neq h^*(X)\right) \\
&\leq \mathbb{P}_X\left(p_{X,h^*(X)} < 0.5\right) \\
&\leq 2\mathbb{E}_X\left[1 - p_{X,h^*(X)}\right] \\
&= 2\mathbb{E}_X\mathbb{P}_{[H]\sim\pi}(H(X) \neq h^*(X)) \\
&= 2\mathbb{E}_{[H]\sim\pi}\mathbb{P}_X(H(X) \neq h^*(X)) \\
&= 2\mathbb{E}_{[H]\sim\pi}[\epsilon_{acc}(H, S)]
\end{aligned}$$

which concludes our proof.

A.9 Proof of Lemma 4.1

Let's consider an ellipsoid on the form

$$z' = \begin{bmatrix} z \\ 0 \end{bmatrix} \qquad P' = \begin{bmatrix} \lambda P & 0 \\ 0 & \nu \end{bmatrix}$$

where $\lambda, \nu > 0$ to ensure P' is positive-definite. In particular, if we assume $1/\lambda + 1/\nu \leq 1$, $\mathcal{E}'$ can be shown to contain $W_t \times [-1, 1]$:

$$\begin{aligned}
w' = (w, w_{t+3}) \in W_t \times [-1, 1] &\Rightarrow w \in W_t \ \wedge\ |w_{t+3}| \leq 1 \\
&\Rightarrow w \in \mathcal{E}(z, P) \ \wedge\ |w_{t+3}| \leq 1 \\
&\Rightarrow (w - z)^T P^{-1}(w - z) \leq 1 \ \wedge\ |w_{t+3}| \leq 1 \\
&\Rightarrow \frac{1}{\lambda}(w - z)^T P^{-1}(w - z) + \frac{1}{\nu} w_{t+3}^2 \leq 1 \\
&\Rightarrow w' \in \mathcal{E}(z', P')
\end{aligned}$$

as desired. Finally, since we are interested in $\mathcal{E}'$ of volume as small as possible, and $vol(\mathcal{E}') \propto \sqrt{\lambda^{t+2}\nu}$, we obtain the following minimization problem:

$$\text{minimize } \lambda^{t+2}\nu, \text{ s.t. } \lambda, \nu > 0 \text{ and } \frac{1}{\lambda} + \frac{1}{\nu} \leq 1$$

This problem can be solved analytically by elementary means, giving as result

$$\lambda^* = 1 + \frac{1}{t+2}, \quad \nu^* = t + 3$$

which concludes our proof.

A.10 Proof of Lemma 4.2

We begin by noticing that W can be written as an intersection of half-spaces:

$$W = \left(\bigcap_{1 \le i \le m} H(a_i, 0) \right) \cap \left(\bigcap_{\|w\|=1} H(w, 1) \right)$$

Thus, checking if $\gamma\mathcal{E} \subset W$ can be reduced to the problem of verifying if $\gamma\mathcal{E} \subset H(a, b)$, for some half-space $H(a, b)$. However, it is known [4, Section 4] that $\gamma\mathcal{E} \subset H(a, b)$ if, and only if, $\alpha(\mathcal{E}, H) \le -\gamma$. By applying this result, we can conclude that

$$\gamma\mathcal{E} \subset W \iff \max_{i \le i \le m} \alpha_i \le -\gamma \text{ and } \max_{\|w\|=1} \alpha_w \le -\gamma$$

where $\alpha_i = \alpha(\mathcal{E}, H(a_i, 0)) = z^T a_i / \sqrt{a_i^T P a_i}$ and $\alpha_w = \alpha(\mathcal{E}, H(w, 1)) = (z^T w - 1)/\sqrt{w^T P w}$, which concludes the proof.

A.11 Proof of Theorem 5.5

First, by the assumption of independent labeling among subspaces we have

$$\begin{aligned} p_{x,y} &= \mathbb{P}(H(x) = y \mid [H] \in \mathcal{V}_f) \\ &= \mathbb{P}(h^s(x^s) = y^s,\ \forall s \mid [h^s] \in \mathcal{V}^s,\ \forall s) \\ &= \prod_s \mathbb{P}(h^s(x^s) = y^s \mid [h^s] \in \mathcal{V}^s) \\ &= \prod_s p^s_{x,y^s} \end{aligned}$$

Therefore, we obtain

$$\begin{aligned} \sum_{y \in \mathcal{Y}_F} p^2_{x,y} &= \sum_{y_1=\pm} \cdots \sum_{y_S=\pm} \prod_s (p^s_{x,y^s})^2 \\ &= \prod_s \sum_{y^s=\pm} (p^s_{x,y^s})^2 \\ &= \prod_s \left((p^s_{x,+})^2 + (p^s_{x,-})^2 \right) \\ &= \prod_s (1 - 2p^s_{x,+}p^s_{x,-}) \end{aligned}$$

A.12 Proof of Theorem 5.6

For $p = (p_1, \cdots, p_S)$, let's define $f(p) = 1 - \prod_s (1 - 2p_s(1 - p_s))$ and $g(p) = \sum_s 2p_s(1 - p_s)$. Additionally, set $x^*_f = \arg\max_x f(p(x))$ and $x^*_g = \arg\max_x g(p(x))$, where $p(x) = (p_{x^1,+}, \cdots, p_{x^S,+})$. We wish to show that $f(x^*_g) \ge \frac{1}{S} f(x^*_f)$, from which our the main result follows from Theorem 13 of Golovin and Krause [14].

It suffices to show $\frac{1}{S} g(p) \le f(p) \le g(p)$ for $p \in [0, 1]^S$, since then

$$f(x^*_g) \ge \frac{1}{S} g(x^*_g) \ge \frac{1}{S} g(x^*_f) \ge \frac{1}{S} f(x^*_f)$$

Now, set $q_s = 1-2p_s(1-p_s) \in [\frac{1}{2}, 1]$, from which we obtain $f(q) = 1-\prod_s q_s$ and $g(q) = S-\sum_s q_s$. By the arithmetic-geometric inequality we have

$$\prod_s q_s \leq \left(\prod_s q_s\right)^{1/S} \leq \frac{1}{S}\sum_s q_s,$$

which gives $f(q) \geq g(q)/S$.

For the second inequality, we proceed by induction on S. $S = 1$ is trivial. Now, assume that the inequality holds for S. It is enough to show that the function

$$h(t) = t\prod_{s=1}^{S} q_s - t - \sum_{s=1}^{S} q_s + S$$

is non-negative for $0 \leq t \leq 1$. Since $h'(t) = \prod_{s=1}^{S} q_s - 1 \leq 0$, $h(t)$ is a decreasing function. However, $h(1) = \prod_{s=1}^{S} q_s - 1 - \sum_{s=1}^{S} q_s + S \geq 0$ by the induction hypothesis, which concludes the proof.

B HIT-AND-RUN IMPLEMENTATION

Let $W \subset \mathbb{R}^n$ be a non-empty bounded convex set. Hit-and-run is a randomized algorithm for generating a uniformly random sample from any convex body. More precisely, this algorithm generates a Markov Chain inside W that converges to the uniform distribution. This Markov Chain can start at any interior point $w_0 \in W$ and then proceeds by iteratively performing the following two steps:

(1) *Hit step*: Generate a random line L passing through w_t and compute its intersection with W
(2) *Run step*: Set w_{t+1} as a random point on the line segment $L \cap W$.

Now, we focus on how to implement this algorithm for convex bodies of the form $W = P \cap B_n$, where P is a polytope $P = \{w \in \mathbb{R}^n : Aw \leq 0\}$ and B_n is unit ball.

The first step is to generate a random line going through w_t. This problem is equivalent to sampling a vector D uniformly over the unit sphere, which can be easily done through Marsaglia's Algorithm [26]: simply sample $D \sim \mathcal{N}(0, I_d)$ and normalize the resulting vector. Once this step is completed, the random line is given by $L = \{w_t + sD,\ s \in \mathbb{R}\}$.

Next, the Hit step requires computing the intersection between L and W, which can be done via simple calculations. First, the intersection with the polytope P is given by

$$\begin{aligned} w_t + sD \in P &\iff A(w_t + sD) \leq 0 \\ &\iff sAD \leq -Aw_t \\ &\iff sa_i^T D \leq -a_i^T w_t, \text{ for each row } a_i \text{ of } A \\ &\iff \max_{a_i^T D<0} -\frac{a_i^T w_t}{a_i^T D} \leq s \leq \min_{a_i^T D>0} -\frac{a_i^T w_t}{a_i^T D} \end{aligned}$$

Next, we consider the intersection with the unit ball B_n. By keeping in mind that $\|D\| = 1$, we have that

$$\begin{aligned} w_t + sD \in B_n &\iff \|w_t + sD\|^2 \leq 1 \\ &\iff s^2 + 2(w_t^T D)s + \|w_t\|^2 - 1 \leq 0 \\ &\iff -w_t^T D - \sqrt{\Delta} \leq s \leq -w_t^T D + \sqrt{\Delta} \end{aligned}$$

where $\Delta = (w_t^T D)^2 - \|w_t\|^2 + 1$. We note that Δ is always positive since $w_t \in B_n$.

Finally, we can conclude that the intersection $L \cap W$ is simply given by $\{w_t + sD,\ s \in [s_{min}, s_{max}]\}$, where $[s_{min}, s_{max}]$ is the intersection of the above two intervals computed. In particular, the Run

Algorithm 5 Hit-and-Run algorithm

Input: convex body $W = \{w \in \mathbb{R}^n : \|w\| \leq 1 \text{ and } a_i^T w \leq 0,\ 1 \leq i \leq m\}$, any $w_0 \in W$, chain length N, inverse rounding transformation T^{-1}
Output: The Hit-and-Run chain $\{w_1, w_2, \ldots, w_N\}$

$samples \leftarrow \{\}$
for t from 0 to $N-1$ **do**
  $D \leftarrow T^{-1}\mathcal{N}(0, I_n)$
  $D \leftarrow D/\|D\|$
  $L_{pol}, U_{pol} \leftarrow \max_{a_i^T D<0} -\frac{a_i^T w_t}{a_i^T D}, \min_{a_i^T D>0} -\frac{a_i^T w_t}{a_i^T D}$
  $\Delta \leftarrow (w_t^T D)^2 - \|w_t\|^2 + 1$
  $L_{ball}, U_{ball} \leftarrow -w_t^T D - \sqrt{\Delta}, -w_t^T D + \sqrt{\Delta}$
  $L, U \leftarrow \max(L_{pol}, L_{ball}), \min(U_{pol}, U_{ball})$
  $s \leftarrow Unif([L, U])$
  $w_{t+1} \leftarrow w_t + sD$
  $samples \leftarrow samples \cup \{w_{t+1}\}$
end for
return $samples$

step can now be easily implemented by sampling S uniformly over $[s_{min}, s_{max}]$ and then setting $w_{t+1} = w_t + SD$.

One last point to consider is how the rounding transformation T affects the hit-and-run chain generation. Let $w_0 \in W$ be the chain's starting point, and let's define $w_0' = T(w_0) \in T(W)$. The usual hit-and-run algorithm over $T(W)$ gives rise to a chain $\{w_t'\}$, which is incrementally defined by $w_{t+1}' = w_t' + s_{t+1}D_{t+1}$. By applying T^{-1} on the previous equation, and setting $w_t = T^{-1}(w_t')$, we obtain a revised version of the hit-and-run update rule:

$$w_{t+1} = w_t + s_{t+1}T^{-1}D_{t+1}$$

In other words, the only necessary change is to multiply the random direction D by T^{-1}. Refer to Algorithm 5 for a pseudo-code of this entire process.

C ROUNDING IMPLEMENTATION

This appendix gives more details on how to implement the optimized rounding algorithm described in Algorithm 3. In particular, we describe how to implement the *minimum_volume_ellipsoid* routine and also how the rounding transformation T can be computed once we have the final rounding ellipsoid.

First, we consider the *minimum_volume_ellipsoid*$(\mathcal{E}, H)$ routine, as described in Goldfarb and Todd [13]. This function computes the minimum volume ellipsoid containing $\mathcal{E} \cap H$, where $\mathcal{E}(z, P) = \{w \in \mathbb{R}^n : (w-z)^T P^{-1}(w-z) \leq 1\}$ is an ellipsoid and $H(a, b) = \{w \in \mathbb{R}^n : a^T w \leq b\}$ a cutting half-space. By defining $\alpha(\mathcal{E}, H) = (a^T z - b)/\sqrt{z^T P z}$, this computation can be separated into 3 special cases:

(1) $\alpha(\mathcal{E}, H) \leq -\frac{1}{n}$: the minimum volume ellipsoid is $\mathcal{E}$ itself.
(2) $\alpha(\mathcal{E}, H) \geq 1$: $\mathcal{E} \cap H$ is empty, and the minimum volume ellipsoid does not exist.
(3) $-\frac{1}{n} < \alpha(\mathcal{E}, H) < 1$: the minimum volume ellipsoid exists and is different from $\mathcal{E}$. It can be computed via Algorithm 6.

In the particular case of our rounding algorithm, only the third case above has to be considered. The first case never happens since at every iteration we pick H satisfying $\alpha(\mathcal{E}, H) > -\gamma > -1/n$; as for the second case, it cannot happen because H is chosen to satisfy $W \subset \mathcal{E} \cap H$, which guarantees $\mathcal{E} \cap H \neq \emptyset$. Thus, as far as the rounding algorithm is concerned, we can directly apply Algorithm 6.

Algorithm 6 Minimum Volume Ellipsoid [13, Section 4]

Input: ellipsoid $\mathcal{E}(z, P = LDL^T)$ in $\mathbb{R}^n$, cutting half-space $H(a, b)$
Output: The minimum volume ellipsoid containing $\mathcal{E} \cap H$

$\hat{a} \leftarrow L^T a$
$\eta \leftarrow \sqrt{\sum_i d_i \hat{a}_i^2}$
$\alpha \leftarrow (a^T z - b)/\eta$
$p \leftarrow \frac{1}{\eta} D\hat{a}$
$\tau \leftarrow (1 - n\alpha)/(n+1)$
$z' \leftarrow z + \tau Lp$
$\delta \leftarrow n^2(1-\alpha^2)/(n^2-1)$
$\sigma \leftarrow 2\tau/(1-\alpha)$
$t_{n+1} \leftarrow \frac{n-1}{n+1}\frac{1-\alpha}{1+\alpha}$
for $j = n, n-1, \ldots, 1$ **do**
  $t_j \leftarrow t_{j+1} + \sigma p_j^2/d_j$
  $d'_j \leftarrow \delta d_j t_{j+1}/t_j$
  $\xi_j \leftarrow -\sigma p_j/(d_j t_{j+1})$
  $v_j^{(j)} \leftarrow p_j$
  for $r = j+1, \ldots, n$ **do**
    $L'_{rj} \leftarrow L_{rj} + \beta_j v_r^{(j+1)}$
    $v_r^{(j)} \leftarrow v_r^{(j+1)} + p_j L_{rj}$
  end for
end for
return $\mathcal{E}(z', P' = L'D'L'^T)$

Another point to note is that Algorithm 6 only requires the LDL^T decomposition of the scaling matrices P as input and it returns the same decomposition for the output ellipsoid. Although this algorithm could be implemented by using P directly, it can lead to numerical instabilities when computing $\eta = \sqrt{a^T P a}$. Thus, it is preferred to use the LDL^T decomposition formulation, which allows for a numerically stable implementation [13].

One final point to consider is how to compute the inverse of the rounding transformation T^{-1}. From our discussion in Section 4.3, T is chosen as any linear transformation mapping the rounding ellipsoid $\mathcal{E}(z, P)$ into a ball of unit radius. Now, let's assume that P can be written as $P = JJ^T$, for some invertible matrix J. Then, we have

$$w \in \mathcal{E}(z, P) \iff (w-z)^T P^{-1}(w-z) \leq 1 \iff \|J^{-1}(w-z)\| \leq 1$$

The above equation implies that J^{-1} maps $\mathcal{E}$ into a unit ball, and we can choose $T^{-1} = J$. For example, we could pick $T^{-1} = LD^{1/2}$, where L and D are the factors in the LDL^T decomposition of P.

D USER QUERIES

In this appendix, we provide an overview of all user interest patterns used in our experiments.The 11 SDSS queries and 18 Car queries are:

SDSS 01: $662.5 < rowc < 702.5$ AND $991.5 < colc < 1053.5$

SDSS 02: $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 29^2$

SDSS 03: $190 < ra < 200$ AND $53 < dec < 57$

SDSS 04: $rowv^2 + colv^2 > 0.5^2$

SDSS 05: $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 90^2$ **AND** $(180 < ra < 210$ AND $50 < dec < 60)$

SDSS 06: $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 280^2$ **AND** $(150 < ra < 240$ AND $40 < dec < 70)$ **AND** $rowv^2 + colv^2 > 0.2^2$

SDSS 07: $x_1 > (1.35 + 0.25 * x_2)$ **AND** $x_3 + 2.5 * \log(2 * 3.1415 * petror50_r^2) < 23.3$

SDSS 08: $(dered_r - dered_i) < 2$ **AND** $17.5 \leq cmodelmag_i - extinction_i \leq 19.9$ **AND** $dered_r - dered_i - (dered_g - dered_r)/8 > 0.55$ **AND** $fiber2mag_i < 21.7$ **AND** $devrad_i < 20.$ **AND** $dered_i < 19.86 + 1.6 * ((dered_r - dered_i) - (dered_g - dered_r)/8. - 0.80)$

SDSS 09: $u - g < 0.4$ **AND** $g - r < 0.7$ **AND** $r - i > 0.4$ **AND** $i - z > 0.4$

SDSS 10: $g \leq 22$ **AND** $-0.27 \leq u - g \leq 0.71$ **AND** $-0.24 \leq g - r \leq 0.35$ **AND** $-0.24 \leq r - i \leq 0.57$ **AND** $-0.35 \leq i - z \leq 0.70$

SDSS 11: $(u - g > 2.0$ OR $u > 22.3)$ **AND** $0 \leq i \leq 19$ **AND** $g - r > 1$ **AND** $r - i < 0.08 + 0.42 * (g - r - 0.96)$ OR $g - r > 2.26$ **AND** $i - z < 0.25$

Car 01: $class$ = 'minivan' **AND** $price_msrp \leq 30000$ **AND** $length > 5$ **AND** $length * width > 10.1$ **AND** $fuel_type$ = 'regular unleaded' **AND** ($transmission \neq$ '8-speed shiftable automatic' AND $transmission \neq$ '9-speed shiftable automatic')

Car 02: $price_msrp \leq 22132$ **AND** $basic_year \geq 5$ **AND** $drivetrain_year \geq 10$ **AND** $horsepower > 156$ **AND** $body_type \neq$ 'suv' **AND** $transmission$ = '6-speed shiftable automatic' **AND** $year \geq 2016$ **AND** $fuel_tank_capacity \geq 65$

Car 03: $year \geq 2016$ **AND** $length * height * width \geq 15.0$ **AND** $basic_year \geq 4$ **AND** $class$ = 'full-size car' **AND** $price_msrp < 100000$ **AND** $engine_type$ = 'gas'

Car 04: $body_type$ = 'truck' **AND** $height \geq 1.9$ **AND** $torque \geq 3800$ **AND** $price_msrp \leq 30000$ **AND** $year = 2017$ **AND** $base_engine_size \geq 5$

Car 05: $class$ = 'full-size van' **AND** $body_type$ = 'van' **AND** $engine_type$ = 'gas' **AND** $model$ = 'nv cargo' **AND** $price_msrp < 27000$ **AND** $horsepower < 300$

Car 06: $height < 1.51$ **AND** $drivetrain_year \geq 10$ **AND** $class$ = 'subcompact car' **AND** ($transmission \neq$ '6-speed manual' **AND** $transmission \neq$ '7-speed manual')

Car 07: $class$ = 'mid-size car' **AND** $transmission$ = '6-speed shiftable automatic' **AND** $drivetrain_year > 5$ **AND** $price_msrp < 29000$ **AND** $basic_km \geq 80467$ **AND** $body_type \neq$ 'suv'

Car 08: $length \geq 6$ **AND** $body_type \neq$ 'sedan' **AND** $drive_type \neq$ 'front wheel drive' **AND** $basic_year \geq 4$ **AND** $drivetrain_year \geq 5$ **AND** $price_msrp < 32000$ **AND** $height > 2.5$ **AND** ($fuel_type \neq$ 'premium unleaded (required)' AND $fuel_type \neq$ 'premium unleaded (recommended)')

Car 09: ($body_type$ = 'truck' OR $body_type$ = 'van') **AND** $price_msrp < 30000$ **AND** $height \geq 2.5$ **AND** $length > 6$

Car 10: $body_type$ = 'truck' **AND** $horsepower > 350$ **AND** $drive_type$ = 'four wheel drive' **AND** $engine_type \neq$ 'diesel' **AND** $fuel_tank_capacity > 100$ **AND** $year \geq 2017$ **AND** $suspension \neq$ 'stabilizer bar stabilizer bar' **AND** $price_msrp < 35000$

Car 11: ($body_type$ = 'suv' OR $body_type$ = 'truck') **AND** ($drive_type$ = 'all wheel drive' OR $drive_type$ = 'four wheel drive') **AND** $height > 1.8$ **AND** $price_msrp < 33000$ **AND** $length > 5.9$ **AND** $suspension$ LIKE '%independent%'

Car 12: $price_msrp < 25000$ **AND** $horsepower > 200$ **AND** $year = 2017$ **AND** $length > 5.5$

Car 13: $price_msrp < 23000$ **AND** $basic_year \geq 5$ **AND** $drivetrain_year \geq 10$ **AND** $basic_km \geq 80000$ **AND** $drivetrain_km \geq 100000$ **AND** $engine_type$ IN ('gas', 'hybrid') **AND** $body_type$ IN ('sedan', 'hatchback') **AND** $drive_type$ = 'front wheel drive' **AND** $fuel_tank_capacity \geq 55$ **AND** $year \geq 2017$

Car 14: $price_msrp < 13000$ **AND** $drive_type$ = 'front wheel drive' **AND** $transmission$ LIKE '%manual%' **AND** $length < 4.5$ **AND** $horsepower < 110$

Car 15: $basic_year \geq 4$ **AND** $year \geq 2017$ **AND** $height \leq 1.62$ **AND** $price_msrp < 25000$ **AND** $transmission$ LIKE '%automatic%' **AND** ($class$ NOT LIKE '%pickup%' AND $class$ NOT LIKE '%suv%')

Car 16: $price_msrp < 26000$ **AND** $body_type$ IN ('van', 'truck') **AND** $2 < height < 2.5$ **AND** $basic_year > 3$

Car 17: $horsepower > 150$ **AND** $year = 2017$ **AND** $make$ IN ('hyundai', 'honda') **AND** $length < 4.5$ **AND** $engine_type$ IN ('gas', 'diesel') **AND** $price_msrp < 20000$

Car 18: $price_msrp < 30000$ **AND** $body_type$ IN ('sedan', 'suv') **AND** $engine_type$ = 'hybrid' **AND** $year = 2017$ **AND** $basic_year \geq 5$ **AND** $drivetrain_year \geq 10$